DirectShow 专业著作

DirectShow 开发指南

陆其明 编著
金邦飞 审校

99%

实用性、综合性

- 深入剖析DirectShow应用架构
- 掌握先进的多媒体应用开发技术
- 全面、深刻、通俗易懂

清华大学出版社

- 本书完全忠实于 DirectX SDK 的帮助文档以及基类源代码，并结合作者多年的实践，经过提炼而成；涉及的内容几乎涵盖了在 Windows 平台上使用 DirectShow 进行 C++ 编码的方方面面

- 本书条理清晰，实用性强，适合广大的流媒体应用开发人员，以及对 Windows 平台上多媒体处理感兴趣的编程爱好者学习和参考

ISBN 7-302-07650-2

9 787302 076506 >

定价：38.00 元

封面设计：谭洁红

读者联系电话：(010) 82896448，82896450

网　址：http://www.khp.com.cn

E-mail：khp@khp.com.cn

DirectShow 开发指南

陆其明　编著
金邦飞　审校

清华大学出版社
北京

内 容 简 介

本书以 DirectX SDK 9.0 版为蓝本，内容几乎涵盖了在 Windows 平台上使用 DirectShow 进行 C++编码的方方面面。全书共分 4 个部分。第 1 部分详细介绍了 DirectShow 的基础知识。第 2 部分重点讨论了 Filter 的开发，以及 DirectShow 应用程序的开发，包括目前非常流行的音视频采集、数码摄像机的支持、非线性编辑等应用。第 3 部分深入分析了 DirectShow SDK 提供的部分典型源代码。第 4 部分结合作者个人的一些开发实践，通过案例和开放源码分析，进一步介绍 DirectShow 的实务应用。

本书内容丰富，条理清晰，实用性强。适合广大的流媒体应用开发人员，以及对 Windows 平台上多媒体处理感兴趣的编程爱好者学习、参考。

版权所有，翻印必究。

本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

图书在版编目（CIP）数据

DirectShow 开发指南 / 陆其明编著.
—北京：清华大学出版社，2003
ISBN 7-302-07650-2

Ⅰ. D… Ⅱ. 陆… Ⅲ. 多媒体—软件工具, DirectShow Ⅳ. TP311.56

中国版本图书馆 CIP 数据核字（2003）第 103827 号

出 版 者：清华大学出版社
http://www.tup.com.cn
社总机：010-62770175

地 址：北京清华大学学研大厦
邮 编：100084
客户服务：010-62776969

组稿编辑：科海
文稿编辑：安靖 潘秀燕
封面设计：谭洁红
版式设计：杨月静
印 刷 者：北京市耀华印刷有限公司
发 行 者：新华书店总店北京发行所
开 本：787×1092 1/16 **印张**：23.25 **字数**：565 千字
版 次：2003 年 12 月第 1 版 2004 年 3 月第 2 次印刷
书 号：ISBN 7-302-07650-2/TP・5612
印 数：5001~6000
定 价：38.00 元

前 言

流媒体的处理以其复杂性和技术性而闻名，一向广受业界的关注。特别是伴随着因特网的普及，流媒体在网络上广泛应用，怎样使流媒体的处理变得简单而富有成效逐渐成为了焦点问题。选择一种合适的应用方案，将会事半功倍。此时，微软公司的 DirectShow 给了我们一个不错的选择。

本书以 DirectX SDK 9.0 版为蓝本，内容几乎涵盖了在 Windows 平台上使用 DirectShow 进行 C++编码的方方面面。全书共分 4 个部分。第 1 部分详细介绍了 DirectShow 的基础知识。第 2 部分重点讨论了 Filter 的开发，以及 DirectShow 应用程序的开发，包括目前非常流行的音视频采集、数码摄像机支持、非线性编辑等应用。第 3 部深入分析了 DirectShow SDK 提供的部分典型源代码例子。第 4 部分结合作者个人的一些开发实践，通过案例和开放源码分析，进一步介绍 DirectShow 的实务应用。

促使我写作这本书的原因是多方面的。首先，我自己学习 DirectShow 的时候并不轻松，因为当时除了 DirectX 的帮助文档外，找不到更多的资料，国内市场上也找不到一本有关 DirectShow 的参考书；而且当时国内从事相关开发的公司或个人甚少，难得有交流的机会。其次，因为我在网上发表了很多有关 DirectShow 的文章，很多朋友认为我是这方面的“高手”。（其实我不是所谓的高手，但我很愿意拿我的平生所学来与大家交流。）于是，经常有人向我咨询如何学习 DirectShow，以及要求我协助解决 DirectShow 应用中的很多实际问题。看到 DirectShow 越来越受欢迎，我一边兴奋，一边也萌发了这样一个念头：“是出一本 DirectShow 书的时候了”。

本书提供 DirectX SDK 9.0 的安装程序，以及书中涉及的所有例子的源代码，下载地址为 http://hqtech.nease.net/DSDeveloper/DsBkSource.zip。

本书是集体智慧的结晶。我首先要感谢彭木昌先生、黄重来先生，以及出版社的夏非彼老师、李才应编辑；没有这些朋友的鼓励和帮助，就不会有这本书的问世。本书初稿完成后，还特邀了敏递软件（上海）有限公司的工程部经理金邦飞先生，进行了严格的技术审校。值得一提的是，本书的封面由我的爱人谭洁红设计。感谢她出色的工作，以及长期以来对本人无微不至的关怀。由于时间短促，再加之本人的水平有限，书中的错误以及疏漏之处在所难免，望广大专家、同行批评指正。

陆其明

2003 年 11 月于上海

目　录

第 1 部分　DirectShow 基础知识

第 1 章　系统概述 1

1.1　DirectX 大家族 1
1.2　DirectShow 简介 2
　1.2.1　DirectShow 系统 2
　1.2.2　播放第一个媒体文件 3
1.3　COM 编程基础 4

第 2 章　Filter 原理 8

2.1　Filter 概述 8
2.2　Filter 的注册 9
2.3　Filter 的媒体类型 13
　2.3.1　majortype 13
　2.3.2　subtype 14
　2.3.3　formattype 15
2.4　Filter 的连接 15
　2.4.1　连接过程 15
　2.4.2　智能连接 22
　2.4.3　动态重建技术 26
2.5　Filter 的数据传送 32
　2.5.1　推模式 33
　2.5.2　拉模式 34
2.6　Filter 的状态转换 35
2.7　媒体定位的实现 39
2.8　质量控制的实现 41
2.9　音视频同步解决方案 44
2.10　对硬件的支持 47
2.11　VMR-9 的发布 49
　2.11.1　VMR 的新特性 49
　2.11.2　VMR 的结构 50
　2.11.3　VMR 使用策略 51

第 3 章　DirectX 媒体对象（DMO）..53

3.1　DMO 概述 53
3.2　DMO 的使用 54
　3.2.1　在应用程序中使用 DMO 54
　3.2.2　在 DirectShow 中使用 DMO 60
3.3　DMO 的开发要点 61
　3.3.1　DMO 中的媒体类型 62
　3.3.2　DMO 的 ATL 实现 63
　3.3.3　DMO 的注册 64

第 2 部分　DirectShow 开发与应用

第 4 章　Filter 组件的开发 66

4.1　开发环境的配置 66
4.2　SDK 基类分析 67
　4.2.1　CBaseObject 68
　4.2.2　CUnknown 68
　4.2.3　CBaseFilter 68
　4.2.4　CBasePin 69
　4.2.5　CBaseInputPin 和 CBase-OutputPin 71
　4.2.6　CSource 72
　4.2.7　CSourceStream 73
　4.2.8　CTransformFilter 74
　4.2.9　CTransInPlaceFilter 75
　4.2.10　CVideoTransformFilter 80
　4.2.11　CBaseRenderer 80
　4.2.12　CBaseVideoRenderer 81
　4.2.13　CPullPin 82
　4.2.14　COutputQueue 83

4.2.15 CSourceSeeking........................83
4.2.16 CEnumPins........................84
4.2.17 CEnumMediaTypes........................85
4.2.18 CMemAllocator........................86
4.2.19 CMediaSample........................87
4.2.20 CBaseReferenceClock........................88
4.2.21 CMediaType........................89
4.2.22 CBasePropertyPage........................89
4.3 Filter 项目的功能分析........................90
4.3.1 功能分析的一般过程........................91
4.3.2 字符叠加 Filter 之功能分析........................92
4.4 Filter 的设计........................93
4.4.1 选择一个合适的父类........................93
4.4.2 应用结构设计........................94
4.5 编码实现........................95
4.5.1 Filter 注册信息........................95
4.5.2 框架函数的实现........................98
4.5.3 逻辑控制类的实现........................101
4.5.4 自定义接口的实现........................109
4.5.5 属性页的实现........................115
4.5.6 产权保护........................119
4.6 Filter 的调试........................120
4.7 MFC Filter........................122

第 5 章 DirectShow 应用开发过程...127

5.1 开发环境的配置........................127
5.1.1 需要包含的头文件........................127
5.1.2 需要连接的库文件........................127
5.1.3 VC 的系统编译环境........................128
5.2 一般开发过程........................128
5.3 通用 Filter Graph 构建技术........................130
5.3.1 加入一个指定 CLSID 的 Filter........................130
5.3.2 得到 Filter 上的未连接 Pin........................131
5.3.3 连接两个 Filter........................132
5.3.4 查找 Filter 或 Pin 上的接口........................134
5.3.5 遍历 Filter 链路........................136
5.3.6 成批删除 Filter........................137
5.4 事件交互的实现........................140
5.5 进度条的实现........................143
5.6 Filter 属性页的显示........................145
5.7 系统设备的枚举........................145
5.8 图片的抓取........................148
5.9 一个简单的媒体文件播放器........................152

第 6 章 音频采集........................156

6.1 应用分析........................156
6.1.1 应用方案........................156
6.1.2 开发要点........................159
6.2 实例解剖........................160
6.2.1 实现的功能........................160
6.2.2 实现要点........................161

第 7 章 视频采集........................166

7.1 应用分析........................166
7.1.1 WDM 与 VFW........................167
7.1.2 构建 Filter Graph........................170
7.1.3 模拟电视接收........................172
7.1.4 输入端子的选择........................178
7.1.5 视频参数的设置........................182
7.1.6 热插拔的支持........................185
7.2 实例解剖........................188
7.2.1 实现的功能........................188
7.2.2 实现要点........................189

第 8 章 数码摄像机的支持........................195

8.1 应用分析........................195
8.1.1 磁带的播放........................197
8.1.2 磁带的录像........................203
8.1.3 DV 的采集........................205
8.2 实例解剖........................208
8.2.1 实现的功能........................208
8.2.2 实现要点........................208

第 9 章 非线性编辑 DES........................214

9.1 DES 概述........................214
9.1.1 时间线模型........................214
9.1.2 时间概念........................217

9.1.3 媒体源......219
9.1.4 音、视频效果与过渡......223
9.1.5 输出控制引擎......226
9.1.6 错误日志......230
9.1.7 DES 项目管理......232
9.2 DES 剖析......233
9.3 DES 新特性......238
9.3.1 视频缩放器的定制......238
9.3.2 解码器的选择......238
9.4 实例解剖......240
9.4.1 实现的功能......240
9.4.2 实现要点......241

第 10 章 DVD 播放......247

10.1 DVD 基础知识......247
10.2 应用分析......248
10.2.1 支持 MPEG2......248
10.2.2 DVD 导航器......250
10.3 实例解剖......253
10.3.1 实现的功能......253
10.3.2 实现要点......253

第 11 章 Windows Media 应用......259

11.1 应用分析......259
11.2 实例解剖......261
11.2.1 实现的功能......261
11.2.2 实现要点......262

第 3 部分 SDK 典型源码分析

第 12 章 Source Filter 例子......267

12.1 拉模式例子......267
12.1.1 实现的功能......267
12.1.2 实现要点......268
12.2 推模式例子......275
12.2.1 实现的功能......275
12.2.2 实现要点......276

第 13 章 Transform Filter 例子......281

13.1 Transform 例子......281
13.1.1 实现的功能......281
13.1.2 实现要点......282
13.2 Trans-In-Place 例子......287
13.2.1 实现的功能......287
13.2.2 实现要点......288

第 14 章 Renderer Filter 例子......292

14.1 Video Renderer 例子......292
14.1.1 实现的功能......292
14.1.2 实现要点......293
14.2 基于 CBaseFilter 例子......296
14.2.1 实现的功能......297
14.2.2 实现要点......297

第 15 章 DMO 例子......304

15.1 实现的功能......304
15.2 实现要点......304

第 16 章 枚举例子......313

16.1 系统枚举例子......313
16.1.1 实现的功能......313
16.1.2 实现要点......313
16.2 Filter 映射例子......317
16.2.1 实现的功能......317
16.2.2 实现要点......318
16.3 DMO 枚举例子......322
16.3.1 实现的功能......322
16.3.2 实现要点......323

第 17 章 媒体播放器例子......325

17.1 实现的功能......325
17.2 实现要点......325

第 4 部分 开放源码分析

第 18 章 MPEG 流的网络客户端播放....336

18.1 需求定义....336

18.2 解决方案....336

18.2.1 Windows Socket 网络传输技术....336

18.2.2 DirectShow 技术应用....337

18.2.3 一种双缓冲队列技术....337

18.3 源码分析....338

18.4 问题会诊....347

第 19 章 MPEG2 视频解码器....349

19.1 需求定义....349

19.2 开放源码分析....349

19.3 Filter 设计....355

19.4 Filter 编码....355

第 1 部分　DirectShow 基础知识

第 1 章　系 统 概 述

1.1　DirectX 大家族

DirectX 软件开发包是微软公司提供的一套在 Windows 操作平台上开发高性能图形、声音、输入、输出和网络游戏的编程接口。微软将 DirectX 定义为“硬件设备无关性”，即使用 DirectX 可以用与设备无关的方法提供设备相关的（高）性能。

事实上，DirectX 已经成为一种标准，它可以为应用程序（特别是游戏）开发人员和硬件厂商之间的关系“解耦”。DirectX 标准的建立，可以为硬件开发提供策略，硬件厂商不得不按照这一标准进行产品改进；同时，通过使用 DirectX 所提供的接口，开发人员可以尽情地利用硬件可能带来的高性能，而无需关心硬件的具体执行细节。

另外，DirectX 采用了组件对象模型（COM）标准，因此不同对象的版本可以有不同的接口，这使用 DirectX 开发的程序即使在未来也能得到完全的兼容和支持。

DirectX 是一项卓越的技术。那么，它为什么称为 DirectX 呢？其实也不难理解，Direct 是直接的意思，X 可以代表很多东西，合在一起就是具有共性的一组东西（这个共性就是直接）。DirectX 是一个大家族，并且随着 DirectX 版本的不断更新，家族成员也在不断地发展壮大。

下面是 DirectX 9.0 家族的所有成员：

- DirectX Graphics：集成了以前的 DirectDraw 和 Direct3D 技术。DirectDraw 主要负责 2D 加速，以实现对显卡内存和系统内存的直接操作；Direct3D 主要提供三维绘图硬件接口，它是开发三维 DirectX 游戏的基础。
- DirectInput：主要支持输入服务（包括鼠标、键盘、游戏杆等），同时支持输出设备。
- DirectPlay：主要提供多人网络游戏的通信、组织功能。
- DirectSetup：主要提供自动安装 DirectX 组件的 API 功能。
- DirectMusic：主要支持 MIDI 音乐合成和播放功能。
- DirectSound：主要提供音频捕捉、回放、音效处理、硬件加速、直接设备访问等功能。
- DirectShow：为在 Windows 平台上处理各种格式的媒体文件的回放、音视频采集等高性能要求的多媒体应用，提供了完整的解决方案。
- DirectX Media Objects：DirectShow Filter 的简化模型，提供更方便的流数据处理方案。

其实，DirectShow 开始并不是 DirectX 家族中的一员，它是经过 DirectX 6.0 中的 DirectX Media 发展而来的。DirectShow 集成了 DirectX 家族中其他成员（如 DirectDraw、DirectSound 等）的技术，可以说是 DirectX 中的一位“集大成者”。经过几个版本的发展，DirectShow 架构日趋成熟。而 DirectX Media Objects 是从 DirectX 8.1 的 DirectShow 中分离出来的，成为了另一种高效率的流数据处理解决方案。

下面将介绍 DirectShow 系统，使大家能够对 DirectShow 有一个总体了解。

1.2 DirectShow 简介

为什么需要 DirectShow? DirectShow 到底能够做什么？带着这两个问题，我们先一起来看多媒体应用开发所面临的挑战:

（1）多媒体数据量巨大，应如何保证数据处理的高效性;

（2）如何让音频和视频时刻保持同步;

（3）如何用简单的方法处理复杂的媒体源问题，包括本地文件、计算机网络、广播电视以及其他一些数码产品等;

（4）如何处理各种各样的媒体格式问题，包括 AVI、ASF、MPEG、DV、MOV 等;

（5）如何支持目标系统中不可预知的硬件。

DirectShow 的设计初衷就是尽量要让应用程序开发人员从复杂的数据传输、硬件差异、同步性等工作中解脱出来，总体应用框架和底层工作由 DirectShow 来完成，这样，基于 DirectShow 框架开发多媒体应用程序就会变得非常简单！

1.2.1 DirectShow 系统

如图 1.1 所示，图中央最大的一块即是 DirectShow 系统，虚线以下是 Ring 0 特权级别的硬件设备，虚线以上是 Ring 3 特权级别的应用层。DirectShow 系统位于应用层中。它使用一种叫 Filter Graph 的模型来管理整个数据流的处理过程；参与数据处理的各个功能模块叫做 Filter；各个 Filter 在 Filter Graph 中按一定的顺序连接成一条“流水线”协同工作。

按照功能来分，Filter 大致分为 3 类：Source Filters、Transform Filters 和 Rendering Filters。Source Filters 主要负责获取数据，数据源可以是文件、因特网计算机里的采集卡（WDM 驱动的或 VFW 驱动的）数字摄像机等，然后将数据往下传输；Transform Filters 主要负责数据的格式转换，例如数据流分离/合成、解码/编码等，然后将数据继续往下传输；Rendering Filters 主要负责数据的最终去向——将数据送给显卡、声卡进行多媒体的演示，或者输出到文件进行存储。

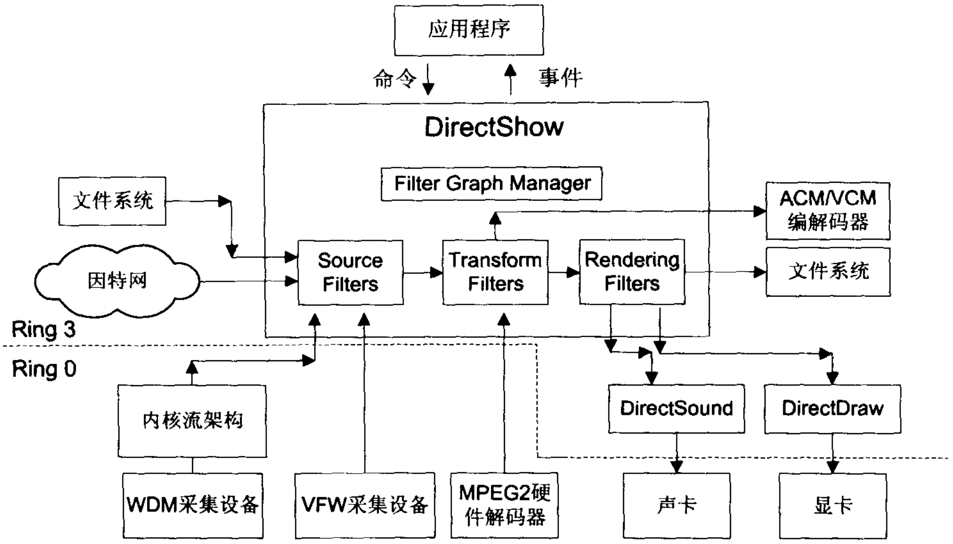

图 1.1　DirectShow 系统

1.2.2　播放第一个媒体文件

DirectShow 提供了大量的 Filter 用以支持最基本的应用。根据实际需要，也可以定制自己的 Filter（如何开发自己的 Filter 将在后续章节逐步介绍）。它的最基本的应用莫过于回放一个媒体文件。如图 1.2 所示是一条典型的 AVI 文件回放的链路。

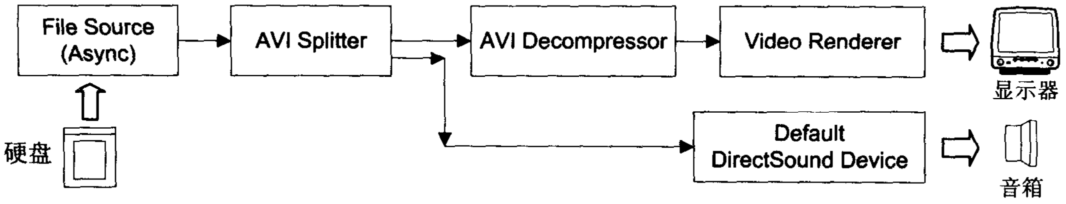

图 1.2　本地 AVI 文件的回放链路

其中，箭头方向即是数据的流向。对照图 1.1 不难发现，File Source（Async）属于 Source Filters，它用于管理硬盘上指定的播放文件，并根据 AVI Splitter 的要求提供数据。AVI Splitter 和 AVI Decompressor 属于 Transform Filters，其中，AVI Splitter 负责向 File Source（Async）索取数据，并将取得数据的音频和视频进行分离，然后分别从各自的输出 Pin 输出；AVI Decompressor 负责视频的解码。Video Renderer 和 Default DirectSound Device 属于 Rendering Filters，其中，Video Renderer 负责向视频窗口输出图像；Default DirectSound Device 负责同步播放声音。

提示

如果你已经安装了 DirectX SDK（可以到微软公司的网站上去下载），运行 SDK 安装目录下的 bin\DXUtils\graphedt.exe，然后选择菜单项 File | Render Media Filter，在弹出的对话框中选择一个媒体文件，可以看到一条回放文件的 Filter 链路。选择菜单项 Graph | Play，开始播放 Filter Graph。

1.3 COM 编程基础

DirectX 采用了 COM 标准。而 DirectShow 是一套完全基于 COM 的应用系统。要想深入学习 DirectShow，掌握一些 COM 编程的基础知识是必不可少的。

对于 DirectShow 应用程序开发人员来说，对 COM 知识的了解并不要求很高。因为 DirectShow 应用程序实际上是一种 COM 组件的客户程序，只是 COM 组件的“使用”问题。这些问题包括如何创建 COM 组件、如何得到组件对象上的接口以及调用接口方法、如何管理组件对象（即需要熟悉 COM 的引用计数机制）等。下面的代码是最一般的步骤。

```
CoInitialize(NULL);             // COM 库初始化
// Do something...
// ...

IUnknown *pUnk    = NULL;
IObject *pObject = NULL;
// 创建组件对象
HRESULT hr = CoCreateInstance(CLSID_Object,
CLSCTX_INPROC_SERVER, NULL, IID_IUnknown, (void**)&pUnk);
if (SUCCEEDED(hr))
{
       // 查询得到组件对象上的接口
     hr = pUnk->QueryInterface(IID_IOjbect,(void**)&pObject);
     if (SUCCEEDED(hr))
     {
       // 调用接口方法
         pObject->SomeMethod();
         pObject->Release();
       }
       pUnk->Release();
}
// ...
CoUninitialize();        // 释放 COM 库使用的资源
```

而对于 Filter 开发人员来说，需要掌握的 COM 知识就要多一点。因为 Filter 本身是一种 COM 组件，开发 Filter 牵涉到了 COM 组件的“实现”问题。

那么，COM 组件到底是什么？COM 本身只是一种规范，而不是实现。但是当使用 C++来实现时，COM 组件就是一个 C++类，而接口都是纯虚类。可以用如下的 C++代码来简单描述一个 COM 组件。

```
class Ifunction
{
public:
virtual Method1(...) = 0;
virtual Method2(...) = 0;
    // ...
};
```

```
class MyObject : public Ifunction
{
public:
virtual Method1(...){...}
virtual Method2(...){...}
    // ...
};
```

其中，Ifunction 就是我们常说的接口，而 MyObject 就是 COM 组件。

COM 规范规定，任何组件或接口都必须从 IUnknown 接口中继承而来。IUnknown 定义了 3 个重要函数，分别是 QueryInterface、AddRef 和 Release。其中，QueryInterface 负责组件对象上的接口查询，AddRef 用于增加引用计数，Release 用于减少引用计数。引用计数是 COM 中的一个非常重要的概念，它很好地解决了组件对象的生命周期问题，即 COM 组件到底在什么时候被销毁，以及由谁来销毁的问题。

提示　COM 组件一般会采用一个"自销毁"的策略。可以看到很多 IUnknown::Release 函数的实现：当引用计数器值为 0 时，组件对象就会调用 delete this 指令。

除了 IUnknown 接口外，还有另外一个重要的接口，即 IClassFactory。COM 组件实际上是一个 C++类，对于组件的外部使用者来说，这个类名一般不可知，那么如何创建这个类的实例？由谁来创建？COM 规范规定，每个组件都必须实现一个与之相对应的类工厂（Class Factory）。类工厂也是一个 COM 组件，它实现了 IClassFactory 接口。在 IClassFactory 的接口函数 CreateInstance 中，才能使用 new 操作生成一个 COM 组件类对象实例。

COM 组件有 3 种类型：进程内组件、本地进程组件和远程组件。Filter 一般是一种进程内组件，以 DLL（动态链接库）的形式提供服务。

接下来看一下怎么来实现一个 COM 组件。如图 1.3 所示是 COM 组件的创建过程。

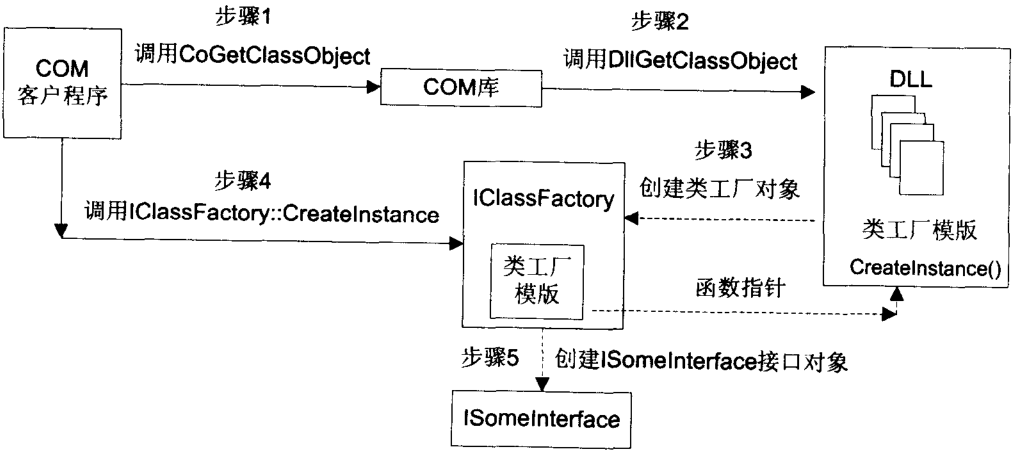

图 1.3　COM 组件的创建过程

下面的代码是 CoCreateInstance 函数实现的伪代码。

```
CoCreateInstance(...)
{
// Do something...
// ...

  IClassFactory *pClassFactory = NULL;
  CoGetClassObject(CLSID_Object, CLSCTX_INPROC_SERVER, NULL,
  IID_IClassFactory, (void **)&pClassFactory);pClassFactory->Create-
  Instance(NULL, IID_IUnknown, (void**)&pUnk);pClassFactory->Release();

  // ...
}
```

每个 COM 组件都使用一个 GUID 来惟一标识。当创建一个 COM 组件时，总是首先通过这个 GUID（上例中为 CLSID_Object）调用 CoGetClassObject 来获得创建这个组件对象的类工厂。然后调用类工厂的接口方法 IClassFactory::CreateInstance，就能真正地创建 CLSID_Object 标识的组件对象了。下面再给出一段 CoGetClassObject 函数以及其他相关函数实现的伪代码。

```
CoGetClassObject(...)
{
// 通过查询注册表 CLSID_Object 得知组件 DLL 文件路径
// 装入 DLL 库（调用 LoadLibrary）
// 使用函数 GetProcAddress(...)得到 DLL 中函数 DllGetClassObject 的函数指针
// 调用 DllGetClassObject 得到类工厂对象指针
}

DllGetClassObject(...)
{
// ...
// 创建类工厂对象
CFactory* pFactory= new CFactory;
// 查询得到 IClassFactory 指针
pFactory->QueryInterface(IID_IClassFactory, (void**)&pClassFactory);
pFactory->Release();
// ...
}

CFactory::CreateInstance(...)
{
// ...
// 创建 CLSID_Object 对应的组件对象
CObject *pObject = new CObject;
pObject->QueryInterface(IID_IUnknown, (void**)&pUnk);
pObject->Release();
// ...
}
```

其中，DllGetClassObject 是在 COM 组件 DLL 中必须实现的一个（导出）函数，它的作用是根据指定的组件 GUID 创建相应的类工厂对象，并返回这个类工厂的 IClassFactory 接

口；CreateInstance 是 IClassFactory 接口的一个接口方法，负责最终创建组件对象实例；CObject 就是我们的 COM 组件类，它实现了 COM 框架以外的真正的组件功能。

一个典型的自注册 COM 组件 DLL 所必需的 5 个（导出）函数如下：

- DllMain：DLL 的入口函数（DirectShow 实现的是 DllEntryPoint）；
- DllGetClassObject：用于获得类工厂指针；
- DllCanUnloadNow：系统空闲时会调用这个函数，以确定是否可以卸载 DLL；
- DllRegisterServer：将 COM 组件注册到注册表中；
- DllUnregisterServer：删除注册表中 COM 组件的注册信息。

提示　其实，SDK 基类源代码中已经实现了 Filter 中的 COM 基本特性。这样，在 SDK 基础上开发 Filter 就没有那么费劲了。了解 COM 组件的创建过程关键是为了理解 Filter 框架的实现。

第 2 章　Filter 原理

2.1　Filter 概 述

Filter 是 DirectShow 中最基本的概念。DirectShow 使用 Filter Graph 来管理 Filter（管理者叫做 Filter Graph Manager）。Filter Graph 是 Filter 的“容器”，而 Filter 是 Filter Graph 中的最小功能模块。

Filter 一般由一个或多个 Pin 组成，Filter 之间通过 Pin 相互连接，构成一条顺序的链路。在第 1 章中曾经提到，Filter 根据实现功能的不同大致可分为 3 类：Source Filters、Transform Filters 和 Rendering Filters。另一种实用的判别方法是根据 Filter 包含的输入 Pin 或者输出 Pin 的数量来判断。

如图 2.1 所示，仅含有输出 Pin，没有输入 Pin 的 Filter 为 Source Filter；既有输入 Pin 又有输出 Pin 的 Filter 为 Transform Filter；仅有输入 Pin，没有输出 Pin 的 Filter 为 Rendering Filter。

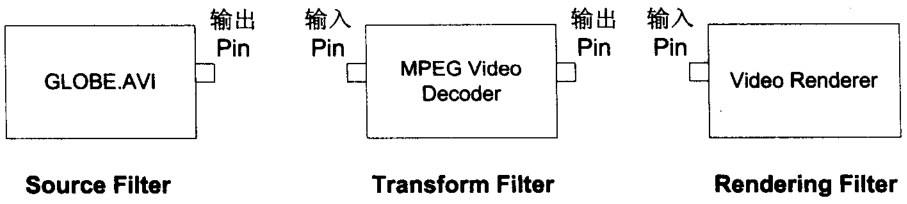

图 2.1　3 种 Filter 类型

Filter 是一种 COM 组件。为了实现在 Filter Graph 中的统一操作，每个 Filter 上都至少实现了 IBaseFilter 接口。实现 Filter 的文件一般是一个 DLL，扩展名可以是.dll，但更多的时候是.ax。跟普通的 COM 组件一样，Filter 的创建是通过 API 函数 CoCreateInstance 来完成的，代码如下：

```
STDAPI CoCreateInstance(
  REFCLSID rclsid,      // Class identifier (CLSID) of the object
  LPUNKNOWN pUnkOuter, // Pointer to controlling IUnknown
  DWORD dwClsContext,  // Context for running executable code
  REFIID riid,         // Reference to the identifier of the interface
  LPVOID * ppv         // Address of output variable that receives
                       // the interface pointer requested in riid
);
```

其中，参数 rclsid 指定要创建的 Filter 的 CLSID；因为绝大多数情况下创建的 Filter 不是被“聚合”（Aggregation）的，所以 pUnkOuter 指定为 NULL；dwClsContext 可以指定为 CLSCTX_INPROC_SERVER，以创建进程内组件对象；riid 在创建 Filter 成功后获得的接口的 ID，一般为 IID_IBaseFilter，也可以是其他特殊的接口；ppv 用于获得接口对象的指针。

提示

Filter 必须加入到 Filter Graph 且接入到工作链路中才能发挥作用。如果想绕过 Filter Graph 而直接使用 Filter 实现的模块功能，微软公司提供了另一种解决方案，就是将 Filter 功能移植成 DirectX 媒体对象（DMO）。这部分内容会在第 3 章详细介绍。

2.2　Filter 的注册

既然 Filter 是一种 COM 组件，使用前就必须先注册。Filter 的注册程序为 regsvr32.exe（位于操作系统目录的 system32 子目录下）。假设现在有一个 Filter 文件，它的完整路径为 C:\DSFilters\myFilter.ax，那么注册这个 Filter 的方法为：在命令行状态下，执行 regsvr32 C:\DSFilters\myFilter.ax。随后会弹出一个对话框，告诉 Filter 注册是否成功。

提示

注销 Filter 也是使用 regsvr32.exe 程序，方法为加命令行参数/u，即执行 regsvr32 /u C:\DSFilters\myFilter.ax。另外一个命令行参数“/s”可以禁止执行 regsvr32.exe 后弹出提示对话框。

给出一个 Filter 的 CLSID，如何判断这个 Filter 是否已经注册？

最简单的办法就是使用 CoCreateInstance 函数创建这个 Filter，然后根据函数的返回值去判断。代码如下：

```
BOOL IsFilterRegistered(CLSID inFilterId)
{
        IBaseFilter * pFilter = NULL;
        if (SUCCEEDED(CoCreateInstance(inFilterId, NULL, CLSCTX_
        INPROC_SERVER,IID_IBaseFilter, (void **)&pFilter)))
        {
                pFilter->Release();
                return TRUE;
        }
        return FALSE;
}
```

一般一个 Filter 项目都会包含一个.def 文件，用于定义 4 个导出函数。代码如下：

```
EXPORTS
        DllGetClassObject    PRIVATE
        DllCanUnloadNow      PRIVATE
        DllRegisterServer    PRIVATE
        DllUnregisterServer  PRIVATE
```

这是 COM 规范的实现。其中，DllGetClassObject 在创建 Filter 对象的时候被调用，根据 CLSID 返回对应的类工厂对象；DllCanUnloadNow 用于判断是否可以从内存中卸载

Filter DLL，即 DLL 中实现的所有 COM 对象是否都已经释放；DllRegisterServer 和 DllUnregisterServer 完成 COM 组件的自注册功能，一般实现代码如下：

```
STDAPI DllRegisterServer()
{
        return AMovieDllRegisterServer2( TRUE );
}
STDAPI DllUnregisterServer()
{
        return AMovieDllRegisterServer2( FALSE );
}
```

提示　AMovieDllRegisterServer2 是 DirectShow SDK 提供的专门用于 Filter 注册的一个 API 函数。可以在 SDK 基类源代码的 dllsetup.cpp 中找到这个函数的实现。

如何在应用程序中注册（或注销）某个 Filter 文件？

只要在应用程序中使用 LoadLibrary 装载这个 Filter 文件，并得到它的导出函数 DllRegisterServer（或 DllUnregisterServer）的入口地址，然后执行它就可以了。下面是注册一个 Filter 的参考代码:

```
BOOL RegisterFilter(const char * inFilterAx)
{
    typedef (WINAPI * REGISTER_FUNC) (void);
    REGISTER_FUNC   MyFunc = NULL;
    HMODULE hModule = ::LoadLibrary(inFilterAx);
    if (hModule)
    {
        MyFunc = (REGISTER_FUNC) GetProcAddress(hModule,
        "DllRegisterServer");
        BOOL pass = (MyFunc != NULL);
        if (pass)
        {
            MyFunc();
        }
        ::FreeLibrary(hModule);
        return pass;
    }
    return FALSE;
}
```

Filter 的注册信息一般包括两部分：基本的 COM 信息和 Filter 特有信息。前者是基本的 COM 组件所必需的信息，没有这部分内容，将无法成功创建 Filter；后者是描述 Filter 的信息（包括 Filter 注册的类型目录、Filter 上 Pin 的数量、支持的媒体类型等），这部分内容会被系统枚举器或者 Filter 映射器访问到，但并不是必需的。

所谓“注册”，也就是向系统注册表的相应位置写入一些数据。Filter 的基本 COM 信

息写入注册表的如下位置：

```
HKEY_CLASSES_ROOT
    CLSID
        Filter CLSID
            REG_SZ: (Default) = Friendly name
            InprocServer32
                REG_SZ: (Default) = File name of the DLL
                REG_SZ: ThreadingModel = Both
```

Filter 特有信息写入注册表的如下位置：

```
HKEY_CLASSES_ROOT
    CLSID
        Category
            Instance
                Filter CLSID
                    REG_SZ: CLSID = Filter CLSID
                    REG_BINARY: Filter Data = Filter information
                    REG_SZ: FriendlyName = Friendly name
```

其中，Category 是为 Filter 注册的类型目录，它是一个 GUID 字符串。

> **提示** 我们自己开发的 Filter 大多数情况下都是注册在 DirectShow Filters 目录下，这个目录的 GUID 为 CLSID_LegacyAmFilterCategory，字符串为{083863F1-70DE-11d0-BD40-00A0C911CE86}。

Filter 的注册目录有很多，可以在如图 2.2 所示的 GraphEdit（即 bin\DXUtils\graphedt.exe）程序中，选择菜单项 Graph | Insert Filters 查看到，如图 2.3 所示。

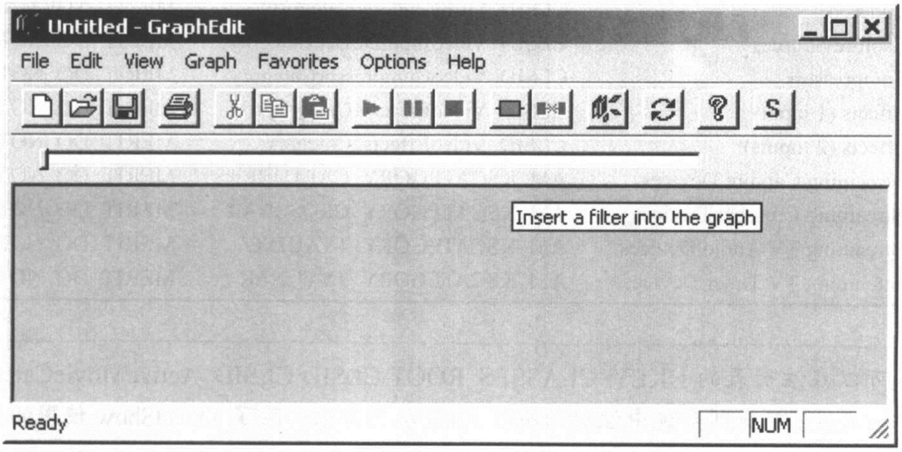

图 2.2　GraphEdit 程序界面

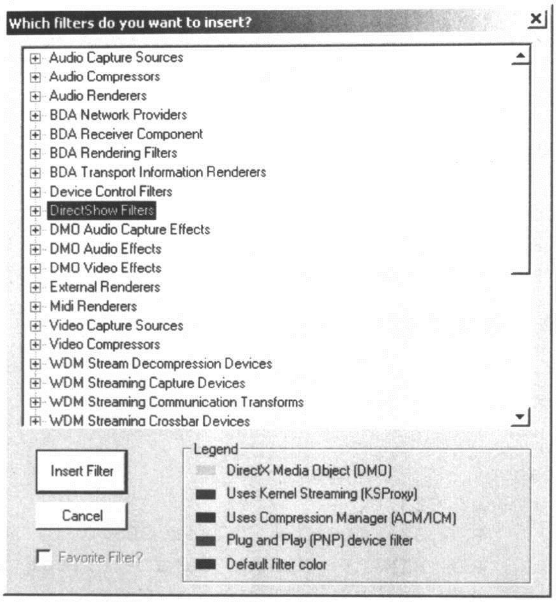

图 2.3　通过 GraphEdit 查看到的 Filter 分类

常用的 Filter 类型目录名与 CLSID 的对照见表 2.1。

表 2.1　常用 Filter 类型目录与 CLSID 对照表

目录名	目录类型 CLSID	目录的 Merit 值
Audio Capture Sources	CLSID_AudioInputDeviceCategory	MERIT_DO_NOT_USE
Audio Compressors	CLSID_AudioCompressorCategory	MERIT_DO_NOT_USE
Audio Renderers	CLSID_AudioRendererCategory	MERIT_NORMAL
DirectShow Filters	CLSID_LegacyAmFilterCategory	MERIT_NORMAL
Midi Renderers	CLSID_MidiRendererCategory	MERIT_NORMAL
Video Capture Sources	CLSID_VideoInputDeviceCategory	MERIT_DO_NOT_USE
Video Compressors	CLSID_VideoCompressorCategory	MERIT_DO_NOT_USE
Video Effects (1 input)	CLSID_VideoEffects1Category	MERIT_DO_NOT_USE
Video Effects (2 inputs)	CLSID_VideoEffects2Category	MERIT_DO_NOT_USE
WDM Streaming Capture Devices	AM_KSCATEGORY_CAPTURE	MERIT_DO_NOT_USE
WDM Streaming Crossbar Devices	AM_KSCATEGORY_CROSSBAR	MERIT_DO_NOT_USE
WDM Streaming TV Audio Devices	AM_KSCATEGORY_TVAUDIO	MERIT_DO_NOT_USE
WDM Streaming TV Tuner Devices	AM_KSCATEGORY_TVTUNER	MERIT_DO_NOT_USE

提示

可以在注册表的 HKEY_CLASSES_ROOT\CLSID\CLSID_ActiveMovieCategories\Instance 下找到系统中注册的所有 Filter 类型目录。除了 DirectShow 使用的标准类型目录外，也可以创建自己的类型目录，只需将自定义的类型目录的 CLSID 在上述位置进行注册即可。在注册表中看到的 CLSID_ActiveMovieCategories 的 GUID 字符串，实际为{da4e3da0-d07d-11d0-bd50-00a0c911ce86}。

2.3　Filter 的媒体类型

Filter 总是要处理一定格式的数据流。那么，用什么来描述这种特定的数据流呢？答案是使用媒体类型（Media Type）。

媒体类型实际上是 DirectShow 定义的一个数据结构 AM_MEDIA_TYPE，代码如下：

```
typedef struct  _MediaType {
    GUID       majortype;
    GUID       subtype;
    BOOL       bFixedSizeSamples;
    BOOL       bTemporalCompression;
    ULONG      lSampleSize;
    GUID       formattype;
    IUnknown   *pUnk;
    ULONG      cbFormat;
    /* [size_is] */ BYTE *pbFormat;
} AM_MEDIA_TYPE;
```

从代码中可以看出，媒体类型主要用 3 部分来描述：majortype（主类型）、subtype（辅助说明类型）和 formattype（格式细节类型）。这 3 部分各自用一个 GUID 来标识。它们的作用分别是：majortype 定性地描述媒体类型，如指定这是一个视频（MEDIATYPE_Video）、音频（MEDIATYPE_Audio）或者字节流（MEDIATYPE_Stream）等；subtype 辅助说明 majortype，指明具体是哪种格式，例如，若 majortype 是视频，subtype 可以进一步指明这是 UYVY（MEDIASUBTYPE_UYVY）、YUY2（MEDIASUBTYPE_YUY2）、RGB24（MEDIASUBTYPE_RGB24）还是 RGB32（MEDIASUBTYPE_RGB32）等，若 majortype 是音频，subtype 可以进一步指明这是 PCM 格式（MEDIASUBTYPE_PCM）还是 AC3 格式（MEDIASUBTYPE_DOLBY_AC3）等；formattype 指定了一种进一步描述格式细节的数据结构类型，格式细节描述的内容主要包括视频图像的大小、帧率，或音频的采样频率、量化精度等参数，这个描述格式细节的数据块指针保存在 pbFormat 成员中。

提示　AM_MEDIA_TYPE 结构中的其他成员，如 bFixedSizeSamples、bTemporalCompression 和 lSampleSize，它们都是可选参数。也就是说，Filter 并不一定定义这些参数，在 Filter 中引用这些参数的值也并不总是“可信”的。这一点对于 Filter 开发人员来说尤为重要。

下面，我们列出了常见的媒体类型。

2.3.1　majortype

主类型 majortype 包括如下媒体类型。

```
MEDIATYPE_Video;                  // 纯视频（数字）
MEDIATYPE_Audio;                  // 纯音频（数字）
MEDIATYPE_AnalogVideo;            // 模拟视频，一般是视频采集卡输入的数据类型
```

```
MEDIATYPE_AnalogAudio;              // 模拟音频，一般是声卡采集输入的数据类型
MEDIATYPE_Text;                     // 文字
MEDIATYPE_Midi;                     // Midi 音乐
MEDIATYPE_Stream;                   // 字节流，如（Pull 模式）文件源的输出数据类型
MEDIATYPE_Interleaved;              // 数码摄像机输入的 DV 数据类型
MEDIATYPE_MPEG1SystemStream;        // MPEG1 的系统流
MEDIATYPE_MPEG2_PACK;               // MPEG2 的数据包
MEDIATYPE_MPEG2_PES;                // MPEG2 分组数据
MEDIATYPE_DVD_ENCRYPTED_PACK;       // DVD 播放用到的媒体类型
MEDIATYPE_DVD_NAVIGATION;
```

2.3.2 subtype

辅助说明类型 subtype 包括如下媒体类型。

```
MEDIASUBTYPE_YUY2;                  // 各种 YUV 的格式
MEDIASUBTYPE_YVYU;
MEDIASUBTYPE_YUYV;
MEDIASUBTYPE_UYVY;
MEDIASUBTYPE_YVU9;
MEDIASUBTYPE_Y411;
MEDIASUBTYPE_RGB4;                  // 各种 RGB 的格式
MEDIASUBTYPE_RGB8;
MEDIASUBTYPE_RGB565;
MEDIASUBTYPE_RGB555;
MEDIASUBTYPE_RGB24;
MEDIASUBTYPE_RGB32;
MEDIASUBTYPE_ARGB32;                // 包含 Alpha 通道的 RGB32
MEDIASUBTYPE_Overlay;               // Overlay Mixer Filter 使用的媒体类型
MEDIASUBTYPE_MPEG1Packet;           // MPEG1 数据包
MEDIASUBTYPE_MPEG1Payload;          // MPEG1 负载（通常表示视频）
MEDIASUBTYPE_MPEG1AudioPayload;   // MPEG1 音频负载
MEDIASUBTYPE_MPEG1System;         // MPEG1 系统流
MEDIASUBTYPE_MPEG1VideoCD;        // VCD
MEDIASUBTYPE_MPEG1Video;          // MPEG1 视频数据
MEDIASUBTYPE_MPEG1Audio;          // MPEG1 音频数据
MEDIASUBTYPE_Avi;                 // AVI
MEDIASUBTYPE_Asf;                 // ASF
MEDIASUBTYPE_QTMovie;             // QuickTime 电影
MEDIASUBTYPE_PCM;                 // PCM
MEDIASUBTYPE_WAVE;                // WAVE 与 PCM 相同
MEDIASUBTYPE_dvsd;                // DV
MEDIASUBTYPE_dvhd;
MEDIASUBTYPE_dvsl;
MEDIASUBTYPE_MPEG2_VIDEO;         // MPEG2 视频
MEDIASUBTYPE_MPEG2_PROGRAM;       // MPEG2 程序流
EDIASUBTYPE_MPEG2_TRANSPORT;      // MPEG2 传输流
MEDIASUBTYPE_MPEG2_AUDIO;         // MPEG2 音频
MEDIASUBTYPE_DOLBY_AC3;           // 杜比的 AC3 音频格式
MEDIASUBTYPE_DVD_SUBPICTURE;      // DVD 的子图片流
MEDIASUBTYPE_DVD_LPCM_AUDIO;      // DVD 的线性 PCM 音频格式
```

```
MEDIASUBTYPE_DVD_NAVIGATION_PCI; // DVD 播放用到的一些媒体类型
MEDIASUBTYPE_DVD_NAVIGATION_DSI;
MEDIASUBTYPE_DVD_NAVIGATION_PROVIDER;
```

2.3.3　formattype

格式细节类型 formattype 对应使用的数据结构代码如下。

```
FORMAT_None
FORMAT_DvInfo           // 描述 DV 数据，使用 DVINFO 数据结构
FORMAT_MPEGVideo        // 描述 MPEG1 数据，使用 MPEG1VIDEOINFO 数据结构
FORMAT_MPEG2Video       // 描述 MPEG2 数据，使用 MPEG2VIDEOINFO 数据结构
FORMAT_VideoInfo        // 描述（一般的）视频数据，使用 VIDEOINFOHEADER 数据结构
FORMAT_VideoInfo2       // 描述（扩展的）视频数据，使用 VIDEOINFOHEADER2 数据结构
FORMAT_WaveFormatEx     // 描述音频数据，使用 WAVEFORMATEX 数据结构
```

当使用一个 AM_MEDIA_TYPE 数据结构描述媒体类型时，如果 majortype、subtype 和 formattype 都指定了一个特定的 GUID 值，那么称这种媒体类型为“完全指定的媒体类型”；这 3 个部分只要有一个指定为 GUID_NULL，则称之为“不完全指定的媒体类型”。GUID_NULL 具有“通配符”的作用。

另外，为了方便使用，DirectShow 提供了另一个 C++类——CMediaType 来操作媒体类型。可以在 SDK 基类源代码的 mtype.cpp 中找到这个类的完整实现。从中可以看到，CMediaType 类是从 AM_MEDIA_TYPE 数据结构中“公共”（public）派生而来的，这样既保留了直接访问数据成员的灵活性，又增加了一些有用的类成员接口，如媒体类型赋值、媒体类型比较、格式数据块内存的自动管理等。

2.4　Filter 的连接

Filter 的连接实际上也就是 Filter 上 Pin 的连接。连接的方向一般总是由上一级 Filter（Upstream Filter）的输出 Pin 指向下一级 Filter（Downstream Filter）的输入 Pin。如图 2.4 所示：

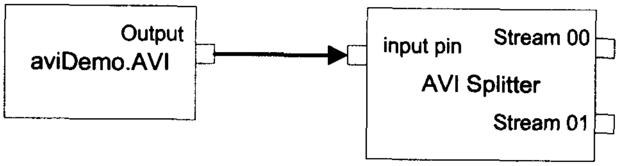

图 2.4　Filter 的连接方向

Pin 的连接实际上是连接双方使用的媒体类型的一个“协商”过程。这个过程很重要，也有点复杂。下面将结合 SDK 基类源代码，详细分析这个过程。

2.4.1　连接过程

Pin 也是一种 COM 组件，而且每个 Pin 上都实现了 IPin 接口。Pin 正是通过这个 IPin 接口来完成连接的。

首先连接 Filter 的是应用程序，一般通过调用接口方法 IFilterGraph::ConnectDirect、

IGraphBuilder::Connect、IGraphBuilder::Render 或者 IGraphBuilder::RenderFile 来实现。

> **提示**
>
> 试图连接的两个 Filter 必须处在同一个 Filter Graph 中。可以调用接口方法 IFilterGraph::AddFilter 将指定的 Filter 加入到 Filter Graph 中。对于 Filter 的加入或者移走，Filter Graph Manager 会调用该 Filter 上的接口方法 Ibase- Filter::JoinFilter-Graph 通知。对于 Filter 开发人员来说，可以重写 CBaseFilter::Join- FilterGraph 函数来捕获这个事件。

Pin 的连接过程如图 2.5 所示。

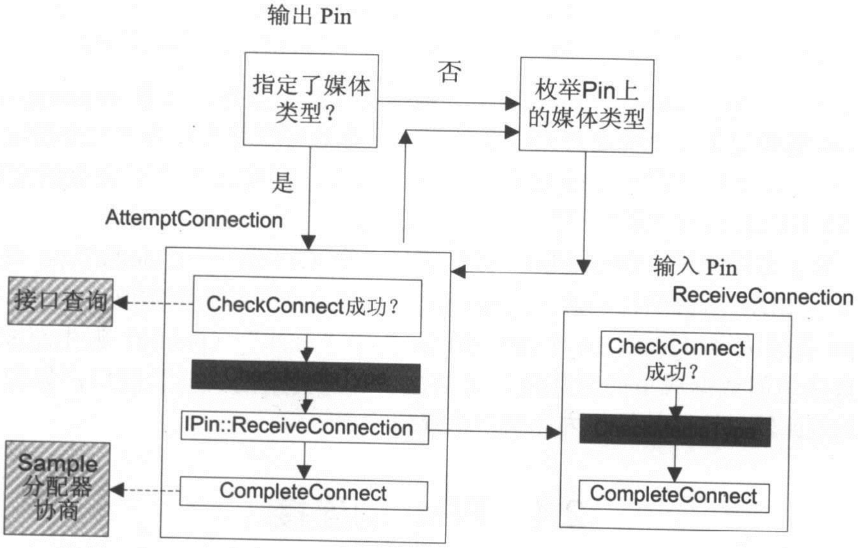

图 2.5　Pin 的连接过程

整个连接过程的步骤大致如下：

（1）Filter Graph Manager 在输出 Pin 上调用 IPin::Connect（带输入 Pin 的指针作为参数）；

（2）如果输出 Pin 接受连接，则调用输入 Pin 上的 IPin::ReceiveConnection；

（3）如果输入 Pin 也接受这次连接，则双方连接成功。

首先来分析第（1）步在基类函数 CBasePin::Connect 中的实现。

```
STDMETHODIMP CBasePin::Connect( IPin * pReceivePin,
    const AM_MEDIA_TYPE *pmt   // optional media type
)
{

    CheckPointer(pReceivePin,E_POINTER);
    ValidateReadPtr(pReceivePin,sizeof(IPin));
    CAutoLock cObjectLock(m_pLock);
    DisplayPinInfo(pReceivePin);
    // 检查该 Pin 是否早已连接
```

```
    if (m_Connected) {
        DbgLog((LOG_TRACE, CONNECT_TRACE_LEVEL, TEXT("Already
              connected")));
        return VFW_E_ALREADY_CONNECTED;
    }
    // 一般 Filter 只能在停止状态下进行连接
    if (!IsStopped() && !m_bCanReconnectWhenActive) {
        return VFW_E_NOT_STOPPED;
    }
    // 开始媒体类型的检查过程，找出一种连接双方都支持的媒体类型
    const CMediaType * ptype = (CMediaType*)pmt;
    HRESULT hr = AgreeMediaType(pReceivePin, ptype);
    if (FAILED(hr)) {
        DbgLog((LOG_TRACE, CONNECT_TRACE_LEVEL, TEXT("Failed to agree
              type")));
        EXECUTE_ASSERT( SUCCEEDED( BreakConnect() ) );
        return hr;
    }
    DbgLog((LOG_TRACE, CONNECT_TRACE_LEVEL, TEXT("Connection succeeded")));
    return NOERROR;
}
```

在上述函数实现中，并没有真正的 Pin 连接过程，只是进行了参数的检查以及状态的检查。还须注意，在连接时，这个函数是从输出 Pin 进入的；Connect 带了两个参数，一个是输出 Pin 试图连接的输入 Pin，另一个是指定的连接用的媒体类型。对于真正的连接过程，还要进一步查看 AgreeMediaType 函数的实现。代码如下：

```
HRESULT CBasePin::AgreeMediaType(IPin *pReceivePin, const CMediaType *pmt)
{
    ASSERT(pReceivePin);
    IEnumMediaTypes *pEnumMediaTypes = NULL;
    // 判断 pmt 是不是一个完全指定的媒体类型
    if ( (pmt != NULL) && (!pmt->IsPartiallySpecified())) {
        // 用这个完全指定的媒体类型进行试连接，如果试连接失败，不再作其他尝试
        return AttemptConnection(pReceivePin, pmt);
    }
    HRESULT hrFailure = VFW_E_NO_ACCEPTABLE_TYPES;
    // 进行 Pin 上支持的媒体类型枚举，开始媒体类型的“协商”过程
    for (int i = 0; i < 2; i++) {
        HRESULT hr;
        if (i == (int)m_bTryMyTypesFirst) {
            hr = pReceivePin->EnumMediaTypes(&pEnumMediaTypes);
        } else {
            hr = EnumMediaTypes(&pEnumMediaTypes);
        }
        if (SUCCEEDED(hr)) {
            ASSERT(pEnumMediaTypes);
            hr = TryMediaTypes(pReceivePin,pmt,pEnumMediaTypes);
            pEnumMediaTypes->Release();
            if (SUCCEEDED(hr)) {
```

```
                return NOERROR;
            } else {
                if ((hr != E_FAIL) &&
                    (hr != E_INVALIDARG) &&
                    (hr != VFW_E_TYPE_NOT_ACCEPTED)) {
                    hrFailure = hr;
                }
            }
        }
    }
    return hrFailure;
}
```

可以从上述 AgreeMediaType 函数的实现中看到，该函数首先判断指定的媒体类型的有效性。如果 pmt 是一个完全指定的媒体类型，那么就以这种媒体类型调用内部函数 AttemptConnection 直接进行试连接。无论这个试连接成功或者失败，连接过程都会停止。事实上，AttemptConnection 完成了第（2）步和第（3）步的处理过程。

但是，如果 pmt 是一个空指针，或者 pmt 包含的是一个不完全指定的媒体类型，那么真正的“协商”过程也就开始了。注意，for 循环的循环次数为 2；输出 Pin 上的成员变量 m_bTryMyTypesFirst 初始值为 false，即为 0。也就是说，连接进程进行到这里，会首先得到输入 Pin 上的媒体类型枚举器的试连接；如果不成功，再得到输出 Pin 上的媒体类型枚举器的试连接。下面再来看一下 TryMediaTypes 函数的实现。

```
HRESULT CBasePin::TryMediaTypes(IPin *pReceivePin, const CMediaType *pmt,
IEnumMediaTypes *pEnum)
{
    // 复位枚举器内部状态
    HRESULT hr = pEnum->Reset();
    if (FAILED(hr)) {
        return hr;
    }
    CMediaType *pMediaType = NULL;
    ULONG ulMediaCount = 0;
    HRESULT hrFailure = S_OK;
    for (;;) {
        hr = pEnum->Next(1, (AM_MEDIA_TYPE**)&pMediaType,&ulMediaCount);
        if (hr != S_OK) {
            if (S_OK == hrFailure) {
                hrFailure = VFW_E_NO_ACCEPTABLE_TYPES;
            }
            return hrFailure;
        }
        ASSERT(ulMediaCount == 1);
        ASSERT(pMediaType);
        // 检查当前枚举得到的媒体类型是否与不完全指定的媒体类型参数匹配
        if ((pmt == NULL) ||
            pMediaType->MatchesPartial(pmt)) {
            // 进行试连接
            hr = AttemptConnection(pReceivePin, pMediaType);
```

```
            if (FAILED(hr) &&
            SUCCEEDED(hrFailure) &&
            (hr != E_FAIL) &&
            (hr != E_INVALIDARG) &&
            (hr != VFW_E_TYPE_NOT_ACCEPTED)) {
                hrFailure = hr;
            }
        } else {
            hr = VFW_E_NO_ACCEPTABLE_TYPES;
        }
        DeleteMediaType(pMediaType);
        if (S_OK == hr) {
            return hr;
        }
    }
}
```

可以看到，当连接进程进入 TryMediaTypes 函数后，会使用媒体类型枚举器枚举 Pin 上提供的所有媒体类型，然后一种一种地进行试连接（调用 AttemptConnection 函数）。在枚举过程中，如果有一种媒体类型试连接成功，那么整个 Pin 连接过程也就宣告成功。如果媒体类型都未获得成功，TryMediaTypes 函数会返回一个错误的值。如果使用输入 Pin 和输出 Pin 的媒体类型枚举器进行试连接都失败了，那么本次 Pin 连接过程宣告彻底失败！

最后，需要看看 AttemptConnection 函数中所做的工作。对于 Filter 开发人员，这部分内容显得尤为重要。因为可以从 AttemptConnection 函数的实现中，清楚地看到一些基类跟 Pin 连接相关的虚函数的调用顺序，这对以后自己写 Filter 时，选择需要重写的基类虚函数具有指导意义。

```
HRESULT CBasePin::AttemptConnection(
    IPin* pReceivePin,
    const CMediaType* pmt
)
{
    // 获取 Filter 对象上的操作权
    ASSERT(CritCheckIn(m_pLock));
    HRESULT hr = CheckConnect(pReceivePin);
    if (FAILED(hr)) {
        DbgLog((LOG_TRACE, CONNECT_TRACE_LEVEL, TEXT("CheckConnect
              failed")));
        EXECUTE_ASSERT( SUCCEEDED( BreakConnect() ) );
        return hr;
    }
    // 一个很有用的调试函数，可以显示媒体类型
    DisplayTypeInfo(pReceivePin, pmt);
    // Pin 上的媒体类型检查
    hr = CheckMediaType(pmt);
    if (hr == NOERROR) {
        m_Connected = pReceivePin;
        m_Connected->AddRef();
        hr = SetMediaType(pmt);    // 在 Pin 上保存媒体类型
```

```
            if (SUCCEEDED(hr)) {
                // 询问连接对方 Pin 是否也能接受当前的媒体类型
                hr = pReceivePin->ReceiveConnection((IPin *)this, pmt);
                if (SUCCEEDED(hr)) {
                    // 完成连接
                    hr = CompleteConnect(pReceivePin);
                    if (SUCCEEDED(hr)) {
                        return hr;
                    } else {
                        DbgLog((LOG_TRACE,
                                CONNECT_TRACE_LEVEL,
                                TEXT("Failed to complete connection")));
                        pReceivePin->Disconnect();
                    }
                }
            }
        } else {
            if (SUCCEEDED(hr) ||
                (hr == E_FAIL) ||
                (hr == E_INVALIDARG)) {
                hr = VFW_E_TYPE_NOT_ACCEPTED;
            }
        }
        EXECUTE_ASSERT( SUCCEEDED( BreakConnect() ) );
        /*  If failed then undo our state */
        if (m_Connected) {
            m_Connected->Release();
            m_Connected = NULL;
        }
        return hr;
    }
```

可以看到，在 AttemptConnection 函数的实现中，首先调用输出 Pin 上的 CheckConnect 函数进行连接检查，例如查看接受连接的 Pin 上是否支持某个特殊的接口等。如果 CheckConnect 失败，会调用 BreakConnect。如果成功，则继续调用输出 Pin 上的 CheckMediaType，进行连接用的媒体类型检查。如果 CheckMediaType 也成功，则会将输入 Pin 对象指针以及当前用的媒体类型保存在输出 Pin 对象中，然后调用输入 Pin 上的 ReceiveConnection 函数。该函数在输入 Pin 上也进行了一系列的检查，检查的内容以及次序与上述输出 Pin 上的类似。如果 ReceiveConnection 调用成功，最后就调用输出 Pin 上的 CompleteConnect 函数。

至此，连接双方 Pin 上使用的媒体类型的“协商”就完成了。那么，这是否意味着 Pin 之间这样就能传送数据了呢？不是！还有些问题没有解决：用于数据传送的内存怎么分配？谁来管理这些内存？实际上，这些“收尾”工作是在上述输出 Pin 上的 CompleteConnect 函数中完成的。下列代码是基类中的源代码实现：

```
HRESULT CBaseOutputPin::CompleteConnect(IPin *pReceivePin)
{
    UNREFERENCED_PARAMETER(pReceivePin);
```

```
    return DecideAllocator(m_pInputPin, &m_pAllocator);
}
```

其中，DecideAllocator 就是完成 Pin 之间数据传送所使用的内存分配器的协商。在 DirectShow 中，数据传送单元叫做 Sample（Sample 是一个 COM 组件，管理一块数据内存）；而 Sample 是由分配器（Allocator，也是一个 COM 组件）来管理的。连接双方的 Pin 必须使用同一个分配器但是这个分配器到底由哪个 Pin 来创建也需要协商。

```
HRESULT CBaseOutputPin::DecideAllocator(IMemInputPin *pPin, IMemAllocator
**ppAlloc)
{
    HRESULT hr = NOERROR;
    *ppAlloc = NULL;
    ALLOCATOR_PROPERTIES prop;
    ZeroMemory(&prop, sizeof(prop));
    // 询问输入 Pin 对分配器的要求
    pPin->GetAllocatorRequirements(&prop);

    if (prop.cbAlign == 0) {
        prop.cbAlign = 1;
    }
    // 询问输入 Pin 是否提供一个分配器
    hr = pPin->GetAllocator(ppAlloc);
    if (SUCCEEDED(hr)) {

        // 决定 Sample 使用的内存大小，以及分配器管理的 Sample 数量
        hr = DecideBufferSize(*ppAlloc, &prop);
        if (SUCCEEDED(hr)) {
            // 通知输入 Pin 最终使用的分配器对象
            hr = pPin->NotifyAllocator(*ppAlloc, FALSE);
            if (SUCCEEDED(hr)) {
                return NOERROR;
            }
        }
    }
    // 如果输入 Pin 上不提供分配器，则必须在输出 Pin 上创建一个分配器
    if (*ppAlloc) {
        (*ppAlloc)->Release();
        *ppAlloc = NULL;
    }
    // 创建一个输出 Pin 上的分配器
    hr = InitAllocator(ppAlloc);
    if (SUCCEEDED(hr)) {

        hr = DecideBufferSize(*ppAlloc, &prop);
        if (SUCCEEDED(hr)) {
            hr = pPin->NotifyAllocator(*ppAlloc, FALSE);
            if (SUCCEEDED(hr)) {
                return NOERROR;
            }
        }
```

```
    }
    if (*ppAlloc) {
        (*ppAlloc)->Release();
        *ppAlloc = NULL;
    }
    return hr;
}
```

可以看到，在 DecideAllocator 函数中，首先询问输入 Pin 上分配器的配置要求。如果输入 Pin 上创建了分配器，则取得这个分配器，然后在输出 Pin 上决定每个 Sample 使用的内存大小、使用的 Sample 数量（在输出 Pin 的 DecideBufferSize 中完成）。如果上述调用都能成功，那么分配器的协商就完成了，即使用输入 Pin 上的分配器。但是，如果上述过程协商不成，输出 Pin 则创建一个自己的分配器，然后，当调用 DecideBufferSize 成功后，调用输入 Pin 的 NotifyAllocator，通知它最终使用的分配器。

提示

当 Pin 上的数据传送内存分配器协商成功后，实际上并没有马上分配 Sample 的内存。实际内存的分配，一般是在 Filter Graph 运行后，调用输出 Pin 上的 Active 函数时进行的，代码如下：

```
HRESULT CBaseOutputPin::Active(void)
{
    if (m_pAllocator == NULL) {
        return VFW_E_NO_ALLOCATOR;
    }
    return m_pAllocator->Commit();
}
```

至此，如果用于数据传送的内存分配器也协商成功了，整个 Pin 的连接过程才算真正完成了。

2.4.2 智能连接

通过上一节的学习，应该对 Pin 的整个连接过程比较清楚了。Filter 通过 Pin 这样的连接，就能“串联”起来，从而构建一个完整的 Filter Graph。

Filter Graph 的构建方法大概有以下几种：

- IFilterGraph::AddFilter：该参数提供一个 Filter 对象，将其加入到 Filter Graph 中
- IFilterGraph::ConnectDirect：该参数提供输出 Pin、输入 Pin 以及媒体类型，进行直接的连接
- IGraphBuilder::AddSourceFilter：该参数提供源文件名，自动将一个 Source Filter 加载到 Filter Graph 中
- IGraphBuilder::Connect：该参数提供输出 pin 和输入 Pin 进行连接，如果连接失败，自动尝试在中间插入必要的格式转换 Filter
- IGraphBuilder::Render：该参数提供输出 Pin，自动加入必要的 Filter 完成剩下部分 Filter Graph 的构建（直到连接到 Rendering Filter）
- IGraphBuilder::RenderFile：该参数提供源文件名，自动加入必要的 Filter 完成这个

文件的回放 Filter Graph 构建

我们看到，后 4 种构建方法都有“自动”的功能。在 DirectShow 中，这种“自动”机制也称为智能连接（Intelligent Connect）。

在介绍智能连接的具体实现过程之前，先要来看一下 Filter 的 Merit 值。这个 Merit 值可以理解成为 Filter 被智能连接所使用的优先级。DirectShow 如下定义了 6 种标准的 Merit 值：

```
enum
{
    MERIT_PREFERRED      = 0x800000,
    MERIT_NORMAL         = 0x600000,
    MERIT_UNLIKELY       = 0x400000,
    MERIT_DO_NOT_USE     = 0x200000,
    MERIT_SW_COMPRESSOR = 0x100000,
    MERIT_HW_COMPRESSOR = 0x100050
};
```

Filter 的 Merit 值只有在大于 MERIT_DO_NOT_USE 的时候，才有可能被智能连接所使用。而在同等条件下（几个 Filter 完成的功能相同），Merit 的值越大，这个 Filter 被使用的机会也就越大。

提示

对于 Filter 开发人员来说，Filter 注册的 Merit 值并不局限于上述 6 个标准的值。原则上可以取任何 DWORD 类型的值。如果不使用标准值，推荐的一种做法是，在标准值上加一个偏移量，如可以注册 Filter 的 Merit 值为 MERIT_DO_NOT_USE + 1。

如何修改系统中已经注册的 Filter 的 Merit 值？

打开注册表，在 Filter 信息的注册部分都会发现一个称为 FilterData 的二进制串类型的键值，这个键值记录的内容就包含 Merit 信息。可以如下定义一个数据结构来操作 FilterData 值开始部分的一块数据：

```
typedef struct
{
     DWORD    dwVersion;     // 版本号
     DWORD    dwMerit;       // Merit 值
     DWORD    dwPinCount;    // Pin 的数量
     DWORD    dwReserved;    // 保留
} FILTER_HEADER;
```

修改 Filter 的 Merit 值的代码如下：

```
BOOL SetFilterMerit(const char * inClsid, DWORD inMerit)
{
        const char *  cRegistryEntry = "CLSID\\{083863F1-70DE-11d0-BD40-
00A0C911CE86}\\Instance\\";
        const long    cMaxLength = 1024 * 16;
        BYTE          filterData[cMaxLength];
        DWORD         actualLength = 0;
```

```
        // 生成 Filter 信息注册部分的注册表入口
        char   szEntry[1000];
        strcpy(szEntry, cRegistryEntry);
        strcat(szEntry, inClsid);
        HKEY hKey   = NULL;
        LONG result = ::RegOpenKeyEx(HKEY_CLASSES_ROOT, szEntry, 0,
                    KEY_ALL_ACCESS, &hKey);
        BOOL pass = (result == ERROR_SUCCESS);
        if (pass)
        {
                // 读取 FilterData 的值
                actualLength = actualLength;
                result = ::RegQueryValueEx(hKey, "FilterData", NULL,
                        NULL, filterData, &actualLength);
                pass   = (result == ERROR_SUCCESS);
        }
        if (pass)
        {
                // 修改 FiterData 中的 Merit 部分，然后写回到注册表
                FILTER_HEADER * filterHeader = (FILTER_HEADER *) filterData;
                filterHeader->dwMerit = inMerit;
                result = ::RegSetValueEx(hKey, "FilterData", NULL,
                        REG_BINARY, filterData, actualLength);
                pass   = (result == ERROR_SUCCESS);
        }
        if (hKey)
        {
                ::RegCloseKey(hKey);
        }
        return pass;
}
```

下面将具体介绍这 4 种智能连接的具体实现过程。

IGraphBuilder::AddSourceFilter

该方法给出一个源文件路径，Filter Graph Manager 能够为这个源文件加载一个相应的 Source Filter。这是一个分析文件路径以及查询注册表的过程。

首先分析文件路径中是否有协议名（如 http、htp 等，Filter Graph Manager 认为冒号前的字符串为协议名，如 myprotocol://myfile.ext 中，myprotocol 就是协议名）。如果有，则在注册表的如下位置寻找这个协议名。

```
HKEY_CLASSES_ROOT
    <protocol>
        Source Filter = <Source filter CLSID>
        Extensions
            <.ext1> = <Source filter CLSID>
            <.ext2> = <Source filter CLSID>
```

找到后，先在 Extensions 子键下匹配扩展名。如果扩展名匹配不成功，则使用 Source Filter 键值指定的 CLSID。默认情况下（注册表中找不到文件路径中的协议名），Filter

Graph Manager 创建一个称为 File Source (URL)的 Filter。

提示　为了区别协议名和驱动器盘符（如 C:\myfile.ext），Filter Graph Manager 不承认单字符的协议名。而且字符串 file:或 file://也不认为是协议名。

如果文件路径中不包含协议名，Filter Graph Manager 就分析这个文件的扩展名，然后到注册表的 HKEY_CLASSES_ROOT\Media Type\Extensions 子键下去匹配扩展名。如果找到了，就根据 Source Filter 键值指定的 CLSID 创建 Filter；找不到，Filter Graph Manager 还有最后一招，就是进行文件的校验字节判断。一些特殊格式的文件校验字节在如下位置注册：

HKEY_CLASSES_ROOT\MediaType\{major type}\{subtype}。

注册表中各个键值结构如下：

```
{e436eb83-524f-11ce-9f53-0020af0ba770}
    {7364696D-0000-0010-8000-00AA00389B71}
        0                           "0,4,,52494646,8,4,,524D4944"
        1                           "0,4,,4D546864"
        Source Filter               "{E436EBB5-524F-11CE-9F53-0020AF0BA770}"
```

打开注册表，可以看到 MediaType 下有一个{e436eb83-524f-11ce-9f53-0020af0ba770}子键（它是 MEDIATYPE_Stream 的字符串形式），其后面又定义了一系列的 Subtype，有{e436eb88-524f-11ce-9f53-0020af0ba770}（即 MEDIASUBTYPE_Avi）、{e436eb84-524f-11ce-9f53-0020af0ba770}（即 MEDIASUBTYPE_MPEG1System）等。在 Subtype 子键下，还可以看到一些子键，其中编号为 0、1、2……的即是定义的校验字节，Source Filter 即为定义的相应的 CLSID。校验字节定义块（Block）一般的格式为：offset,cb,mask,val，意思为在文件头偏移 offset 字节处读取 cb 个字节的数据，将其与 mask 做位与操作，结果等于 val 即表示匹配；如果定义的 mask 为空，可以不做位与操作；如果 offset 为负数，表示从文件尾开始计算偏移；如果一个子键有多个定义块，则所有的定义块都匹配才能算这个子键匹配；如果有多个子键用于校验字节的定义，那么任何一个匹配就表示 Source Filter 的匹配。如果注册表中定义的所有校验字节都不能匹配，Filter Graph Manager 最后就只能默认创建一个称为 File Source (Async.)的 Filter，Media Type 使用{MEDIATYPE_Stream, MEDIASUBTYPE_None}。

IGraphBuilder::RenderFile

该方法给出一个文件名，首先根据与 IGraphBuilder::AddSourceFilter 类似的算法找到并创建正确的 Source Filter。然后，就从该 Source Filter 的各个输出 Pin 开始，进行剩下的智能连接过程。这是一个“递归”过程，直到所有的分支都连到一个 Rendering Filter 上为止。步骤大致为：

（1）如果输出 Pin 支持 IStreamBuilder 接口，则把剩下的工作交给 IstreamBuilder::Render；

（2）使用在 Filter Graph Manager 内部缓冲（Cache）中的 Filter 进行试连接；

（3）使用在当前 Filter Graph 中还没有完全连接的 Filter 进行试连接；

（4）使用 IFilterMapper2::EnumMatchingFilters 搜索注册表。Filter Graph Manager 使用 Merit 值大于 MERIT_DO_NOT_USE 的所有 Filter（Filter 所在的类型目录的 Merit 值也应该大于 MERIT_DO_NOT_USE）进行试连接。在匹配 Mediatype 的前提下，Merit 值越高，该 Filter 被使用的概率就越高。

IGraphBuilder::Render

这个方法从当前 Filter Graph 的某个 Filter 的指定输出 Pin 开始，进行从这个 Pin 往下的一条支路的智能连接。智能连接的算法与上述 RenderFile 的类似。

IGraphBuilder::Connect

这个方法以欲连接的一对输出 Pin 和输入 Pin 作为参数。首先进行这两个 Pin 之间的直接连接。如果不成功，则要插入“中介”Filter。这个“中介”Filter 的选择以及试连接过程就是一个智能连接过程，算法与上述的 RenderFile 类似。

如何让智能连接过程优先使用我们指定的某个 Filter?

在执行智能连接函数之前，可以首先将我们的 Filter 加入到 Filtr Graph 中。或者再提高一个优先级：将 Filter 加入到 Filter Graph Manager 的内部缓冲中。后一种方法可以从 Filter Graph Manager 上获得 IGraphConfig 接口，然后调用其接口方法 AddFilterToCache 来实现。值得注意的是，如果要加入到缓冲中的 Filter 已经在 Filter Graph 中了，则这个 Filter 的所有 Pin 必须都处于断开状态。

2.4.3 动态重建技术

由于下列任何一个原因，我们都需要对已有的 Filter Graph 进行修改。

- 应用程序在播放一段视频的过程中想要插入一个视频效果 Filter;
- Source Filter 在运行过程中改变了数据流的媒体类型，需要接入新的解码 Filter;
- 应用程序想要在 Filter Graph 中加入另外一条视频流。

通常的做法是，先将 Filter Graph 停止，进行修改之后，再重新启动。那么，有没有一种方法能够在保持 Filter Graph 运行状态的同时实现 Filter Graph 的重建呢？答案是肯定的。

Filter Graph 的重建包括如下几种情形：

- 仅仅改变 Filter 之间连接的媒体类型;
- 增加或删除 Filter，重新进行相关 Filter 之间的连接;
- 对一条 Filter 链路（Filter Chain）进行操作。

下面将分别进行讨论。

媒体类型的动态改变

大家知道，当两个 Pin 连接完成后，就会有一个双方商定的媒体类型用来描述以后在这两个连接 Pin 之间传送的数据格式。一般情况下，这个媒体类型在 Filter Graph 运行的

整个过程中是不会改变的。但在一些特殊的场合下，“需要”这种媒体类型的动态改变。DirectShow 是如何支持的呢？下面分不同情况来讨论。

从上往下要求媒体类型改变，但传送数据使用的内存不需增大

如图 2.6 所示，媒体类型的改变由 Filter A 的输出 Pin 发起。首先，Filter A 调用 Filter B 的输入 Pin 上的 IPin::QueryAccept 或 IPinConnection::DynamicQueryAccept，看 Filter B 是否支持新的媒体类型。如果支持，则 Filter A 调用 IMemAllocator::GetBuffer 得到一个空的 Sample，通过 IMediaSample::SetMediaType 将新的媒体类型与 Sample 相连。然后，Filter A 就可以将 Sample 传送给 Filter B。当 Filter B 接收到新的 Sample 时，可以通过调用 IMediaSample::GetMediaType 得到新的媒体类型。

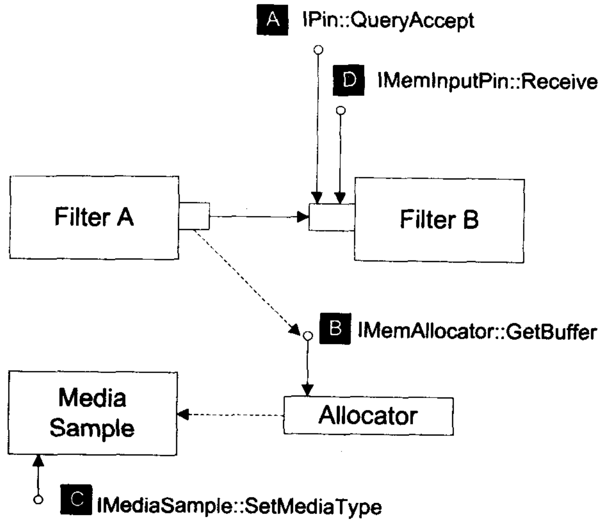

图 2.6　从上往下的媒体类型动态改变

一般来说，Filter B 应该具有对媒体类型改变的应变能力，这是在写自己的 Filter 时应该考虑到的问题。Filter A 一般调用 QueryAccept 来判断 Filter B 的输入 Pin 是否支持新的媒体类型。可以从 CBasePin::QueryAccept 的实现中看到，它仅仅是调用 Pin 上的 CheckMediaType 函数。所以 QueryAccept 返回 S_OK 并不能保证媒体类型能够在 Graph 运行状态下改变成功。另一种可靠的方法是，如果 Filter B 的输入 Pin 支持 IPinConnection 接口，那么调用这个接口的 DynamicQueryAccept 方法。

对于 Filter 开发人员来说，还需注意一下 CBaseInputPin::Receive 的实现，代码如下：

```
STDMETHODIMP CBaseInputPin::Receive(IMediaSample *pSample)
{
    CheckPointer(pSample,E_POINTER);
    ValidateReadPtr(pSample,sizeof(IMediaSample));
    ASSERT(pSample);
    // 检查 Filter 内部状态，以及数据流传送状态
    HRESULT hr = CheckStreaming();
    if (S_OK != hr) {
        return hr;
```

```
    }
    // 查询 IMediaSample2 接口，获取输入 Sample 上的属性
    IMediaSample2 *pSample2;
    if (SUCCEEDED(pSample->QueryInterface(IID_IMediaSample2, (void
**)&pSample2))) {
        hr = pSample2->GetProperties(sizeof(m_SampleProps),
(PBYTE)&m_SampleProps);
        pSample2->Release();
        if (FAILED(hr)) {
            return hr;
        }
    } else {
        // 使用 ImediaSample 接口获取输入 Sample 上的属性
        m_SampleProps.cbData = sizeof(m_SampleProps);
        m_SampleProps.dwTypeSpecificFlags = 0;
        m_SampleProps.dwStreamId = AM_STREAM_MEDIA;
        m_SampleProps.dwSampleFlags = 0;
        if (S_OK == pSample->IsDiscontinuity()) {
            m_SampleProps.dwSampleFlags |= AM_SAMPLE_DATADISCONTINUITY;
        }
        if (S_OK == pSample->IsPreroll()) {
            m_SampleProps.dwSampleFlags |= AM_SAMPLE_PREROLL;
        }
        if (S_OK == pSample->IsSyncPoint()) {
            m_SampleProps.dwSampleFlags |= AM_SAMPLE_SPLICEPOINT;
        }
        if (SUCCEEDED(pSample->GetTime(&m_SampleProps.tStart,
                                        &m_SampleProps.tStop))) {
            m_SampleProps.dwSampleFlags |= AM_SAMPLE_TIMEVALID |
                                            AM_SAMPLE_STOPVALID;
        }
        if (S_OK == pSample->GetMediaType(&m_SampleProps.pMediaType)) {
            m_SampleProps.dwSampleFlags |= AM_SAMPLE_TYPECHANGED;
        }
        pSample->GetPointer(&m_SampleProps.pbBuffer);
        m_SampleProps.lActual = pSample->GetActualDataLength();
        m_SampleProps.cbBuffer = pSample->GetSize();
    }
    // 检测 Sample 上的媒体类型是否改变
    if (!(m_SampleProps.dwSampleFlags & AM_SAMPLE_TYPECHANGED)) {
        return NOERROR;
    }
    // 检查输入 Pin 上是否支持这种新的媒体类型。此时，输入 Pin 应该是支持这种媒体类型
    // 的，因为上一级 Filter 在做出这种媒体类型的动态改变之前，一般会先调用输入 Pin
    // 上的 IPin::QueryAccept 接口方法进行确认
    hr = CheckMediaType((CMediaType *)m_SampleProps.pMediaType);
    if (hr == NOERROR) {
        return NOERROR;
    }
    m_bRunTimeError = TRUE;
```

```
    EndOfStream();
    m_pFilter->NotifyEvent(EC_ERRORABORT,VFW_E_TYPE_NOT_ACCEPTED,0);
    return VFW_E_INVALIDMEDIATYPE;
}
```

我们可以看到，这个函数首先调用 CheckStreaming 来检查 Filter 的运行状态，其次就是读取 Sample 的属性（保存到成员变量 AM_SAMPLE2_PROPERTIES m_SampleProps 中），判断媒体类型是否已经改变。如果媒体类型改变了，会重新调用 CheckMediaType 进行检查。这就是 SDK 基类提供给我们的（也是仅有的）、对媒体类型动态改变的"免疫能力"。其实，对于媒体类型动态改变的支持，还需要处理更多的细节问题（如可能需要重新初始化等），这要因具体的 Filter 任务而异。

从下往上要求媒体类型改变，但传送数据使用的内存不需增大

提示　这里的从下往上要求媒体类型改变有一个前提条件，那就是 Filter B 必须拥有自己的 Sample 分配器。如果 Filter A 与 Filter B 连接双方的 Pin 使用的 Sample 分配器是 Filter A 内部创建的，则 Filter B 将不能支持这种媒体类型的改变。

如图 2.7 所示，媒体类型的改变由 Filter B 的输入 Pin 发起。首先，Filter B 调用 Filter A 的输出 Pin 的 QueryAccept，看是否支持新的媒体类型。如果支持，则 Filter B 内部通过私有方法来改变 Sample 管理器中的下一个空 Sample 的媒体类型（因为这个分配器是 Filter B 创建的，所以 Filter B 有办法对下一个空 Sample 调用 IMediaSample::SetMediaType，而无需通过公有接口方法 IMemAllocator::GetBuffer 先将下一个空 Sample 取出）。接下来当 Filter A 调用 IMemAllocator::GetBuffer 取得一个空 Sample 时，就可以通过 IMediaSample::GetMediaType 来得到新的媒体类型。此时，Filter A 必须根据新的媒体类型产生 Sample 数据，然后再传给 Filter B。最后，Filter B 在接收到 Sample 时，会确认媒体类型的改变。

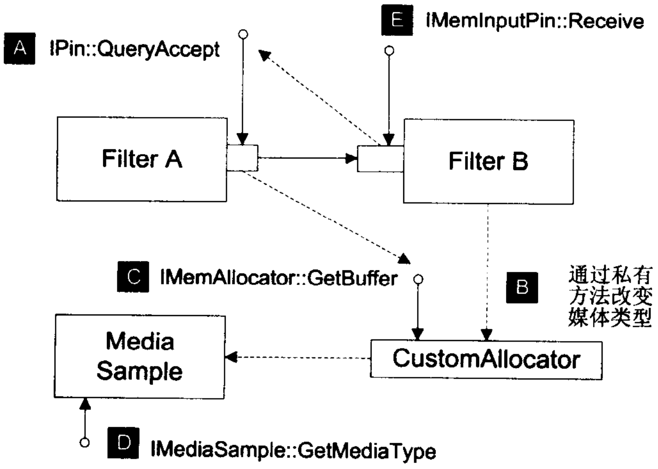

图 2.7　从下往上的媒体类型动态改变

提示

如果 Filter A 能够接受新的媒体类型，那么它也应该有能力恢复到原来的媒体类型。这种媒体类型的动态改变，最常见于传统的 Video Renderer 上。大家可能已经发现，Video Renderer 在连接时用的媒体类型是 RGB，但运行以后有时候却变成了 YUV，它的工作原理就如上所述。Video Renderer 首先使用 RGB 的媒体类型连接，是为了兼顾 GDI 函数的应用。为了高效地画图，当能够使用 DirectDraw 时，Video Renderer 会要求上一级 Filter 输出 YUV 格式；DirectDraw 表面（Surface）也有可能中途丢失，Video Renderer 会再次要求上一级 Filter 恢复 RGB 格式，以便使用 GDI 函数画图。

输出 Pin 往下要求新的媒体类型，而且需要使用更大的内存以传送新的 Sample

在上述两种情形中，虽然媒体类型改变了，但是 Sample 使用的内存大小并不需要扩大。当新的媒体类型要求更大的内存用于传送数据时，上一级的 Filter 必须进行如下处理：

（1）调用下一级 Filter 的输入 Pin 上的 IPin::ReceiveConnection。如果成功返回，且需要改变 Sample 内存大小，则在输出 Pin 上调用 IMemAllocator::SetProperties，以使用新的内存大小进行 Sample 内存的重新分配；

（2）调用输入 Pin 上的 IMemInputPin::NotifyAllocator 通知使用新的 Sample 分配器。

对于 Filter 开发人员来说，还需要注意的一点是，在做出上述改变之前，确信已将所有旧媒体类型的 Sample 都发送出去了。

提示

一些 MPEG 视频解码 Filter 就是使用这种机制来实现输出 MPEG-1 与 MPEG-2 之间的切换，或者视频图像大小的改变的。

动态删除或增加 Filter

如图 2.8 所示，假设我们想将 Filter 2 动态移走。

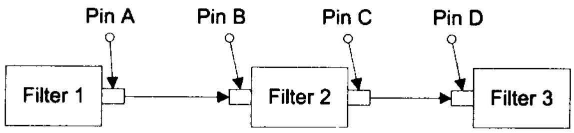

图 2.8　动态删除或增加 Filter

提示

要实现动态删除或增加 Filter，有两个必要条件：（1）Filter 3 (Pin D)必须支持 IPinConnection 接口（这个接口保证 Filter 在非停止状态下也能进行 Pin 的重连）；（2）Filter 在重连的时候不允许数据的传输，所以要阻塞数据线程。如果“重连”是由 Filter 1 发起的（在 Filter 内部完成），那么 Filter 1 要同步这个数据发送线程；如果“重连”是由应用程序完成的，则要求 Filter 1（Pin A）实现 IPinFlowControl 接口。

动态重连的一般步骤如下：

（1）在 Filter 1 上阻塞数据流线程。IPinFlowControl::Block 可以工作在同步和异步两种模式下。不要在应用程序主线程下使用该 Block 函数的同步模式，因为这样可能会引起线程死锁（Dead-lock）。应该另外使用一个工作者线程，或者使用 Block 函数的异步模式。代码如下：

```
// 创建一个同步事件对象
HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
if (hEvent != NULL)
{
    // 阻塞数据流线程
    hr = pFlowControl->Block(AM_PIN_FLOW_CONTROL_BLOCK, hEvent);
    if (SUCCEEDED(hr))
    {
        // 等待完成阻塞工作
        DWORD dwRes = WaitForSingleObject(hEvent, dwMilliseconds);
    }
}
```

（2）重连 Pin A 和 Pin D，必要时插入新的 Filter。这里 Pin 的重连可以使用 IGraph-Config::Reconnect 或 IGraphConfig::Reconfigure（IGraphConfig 接口可以从 Filter Graph Manager 上获得）。Reconnect 比 Reconfigure 使用起来要简单，它主要做如下几件事：将 Filter 2 置于停止状态，然后将其移走；加入新的 Filter；重新连接相关的各个 Pin；将新加入的 Filter 置于暂停或运行状态，以使其与所在的 Filter Graph 同步。代码如下：

```
pGraph->AddFilter(pNewFilter, L"New Filter for the Graph");
pConfig->Reconnect(
    pPinA,      // Reconnect this output pin...
    pPinD,      // ... to this input pin
    pMediaType, // Use this media type
    pNewFilter, // Connect them through this filter
    NULL,
0);
```

在实际应用中，如果大家觉得 Reconnect 不够灵活，还可以改用 Reconfigure。使用 Reconfigure 方法时，必须在应用程序里实现 IGraphConfigCallback 接口；在调用 Reconfigure 之前，还必须依次调用 Filter 3（Pin D）上的 IPinConnection::NotifyEndOfStream 和 Filter 2（Pin B）上的 IPin::EndOfStream，以使得还没处理完的数据全部发送下去（这些处理 IGraphConfig::Reconnect 会自动完成）。

（3）再次启动 Filter 1（Pin A）上的数据发送线程。只需调用 IPinFlowControl::Block 即可，代码如下：

```
pFlowControl->Block(0, NULL);
```

Filter Chain 操作

首先要弄明白什么是 Filter Chain。如图 2.9 所示，Filter Chain 是相互连接着的一条 Filter 链路，并且链路中的每个 Filter 至多有一个处于“已连接”状态的输入 Pin，至多有

一个处于“已连接”状态的输出 Pin。这条 Filter 链路中的数据流不依赖链路外的其他 Filter。图中的 A–B，C–D，F–G–H，F–G，G–H 都是 Filter Chain，而任何含有 E 的都不是 Filter Chain。

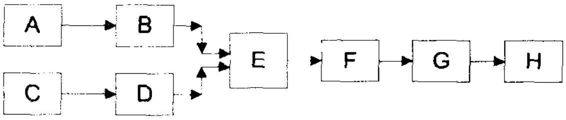

图 2.9 Filter Chain 操作

Filter Chain 通过 IFilterChain 接口来进行相关的操作。该接口可以从 Filter Graph Manager 上获得，它的所有接口方法见表 2.2。

表 2.2 IfilterChain 接口方法

接口方法	描述
StartChain	将指定起始 Filter 和结束 Filter 的 Filter Chain 置于运行状态
StopChain	将指定起始 Filter 和结束 Filter 的 Filter Chain 置于停止状态
RemoveChain	将指定起始 Filter 和结束 Filter 的 Filter Chain 中的所有 Filter 从 Filter Graph 中删除
PauseChain	将指定起始 Filter 和结束 Filter 的 Filter Chain 置于暂停状态

提示

没有一个专门的接口方法可以加入或者连接一个 Filter Chain。Filter Chain 的概念是通过 IFilterChain 的接口方法的两个参数（一个指定起始的 Filter，一个指定结束的 Filter）来体现的。

当 Filter Graph 处于运行状态下，Filter Chain 可以在运行和停止状态之间切换；当 Filter Graph 处于暂停状态下，Filter Chain 可以在暂停和停止状态之间切换。以上是 Filter Chain 仅有的两种状态转换。

在使用 IFilterChain 接口之前，还需要确认要操作的 Filter 是否支持 Filter Chain 操作，否则很容易引起线程死锁。一般为了使用 Filter Chain 特性，需要特殊设计一些 Filter。那么为什么要在介绍“动态重建技术”这一节时提到 Filter Chain 呢？这是因为接口方法 IFilterChain::RemoveChain 能够一次移走好几个 Filter；而且，移走一个 Filter Chain 的步骤与上述第（2）种情形类似。

2.5 Filter 的数据传送

Filter 之间的成功连接为数据传送做好了准备。前面已经提及，Filter 之间是以 Sample 的形式来传送数据的。Sample 是一个封装了一定大小数据内存的 COM 组件。那么，数据到底是怎么从上一级 Filter 传送到下一级 Filter 的呢？

大家知道，每个 Pin 上都实现了 IPin 接口。但这个接口主要是用于 Pin 连接的，而不是用于数据传送的。真正用于数据传送的一般是输入 Pin 上实现的 IMemInputPin 接口（调用其接口方法 IMemInputPin::Receive）。

提示

这里加上了“一般”二字，是因为 IMemInputPin 并不是数据传送的惟一方法。实际上，应用场合不同，数据传送的方式也有所不同。我们将在下文作详细介绍。

如图 2.10 所示是 Filter 之间数据传送的示意图。

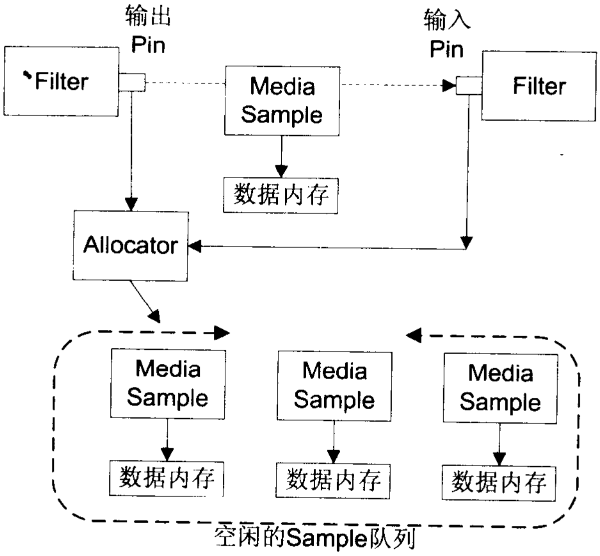

图 2.10　Filter 之间的数据传送过程

可以看到，连接着的双方 Pin 拥有同一个 Allocator（即 Sample 分配器）；Allocator 创建、管理一个或多个 Sample。数据传送时，上一级 Filter 从输出 Pin 的 Allocator 中（调用 IMemAllocator::GetBuffer）得到一个空闲 Sample，然后得到 Sample 的数据内存地址，将数据放入其中。最后，再将这个 Sample 传送给下一级 Filter 的输入 Pin。

提示

数据从上一级 Filter 的输出 Pin 传送到下一级 Filter 的输入 Pin，并没有进行数据的内存备份，而只是数据到达的“通知”。典型的数据内存备份发生在 Filter 内部，从这个 Filter 的输入 Pin（经过一定的处理后）复制到 Filter 的输出 Pin。

数据传送主要有两种模式：推模式（Push Model）和拉模式（Pull Model）。

2.5.1　推模式

推模式最典型的情况发生在 Live Source（实时源，如采集卡的 Capture Filter 等）中。这种源能够自己产生数据，并且使用专门的线程将这些数据“推”下去。推模式也是 DirectShow 中最常见的一种数据传送方式。如图 2.11 所示，数据从 Capture Pin 出来，调用 AVI Decompressor 的输入 Pin 上的 IMemInputPin::Receive 函数，实现数据从 Capture Filter 到 AVI Decompressor 的传送；然后，在 AVI Decompressor 内部，输入 Pin（在自己的 Receive 函数中）接收到数据后，Filter 将这块数据进行格式转换，再将转换后的数据放

到输出 Pin 的 Sample 中，调用 Null Renderer 的输入 Pin 上的 IMemInputPin::Receive 函数，从而实现数据从 AVI Decompressor 到 Null Renderer 的传送；Null Renderer（在输入 Pin 的 Receive 函数中）接收到数据后进行必要的处理后就返回。至此，数据传送的一个轮回也就完成了。如图 2.12 所示为这个传送过程的示意图。

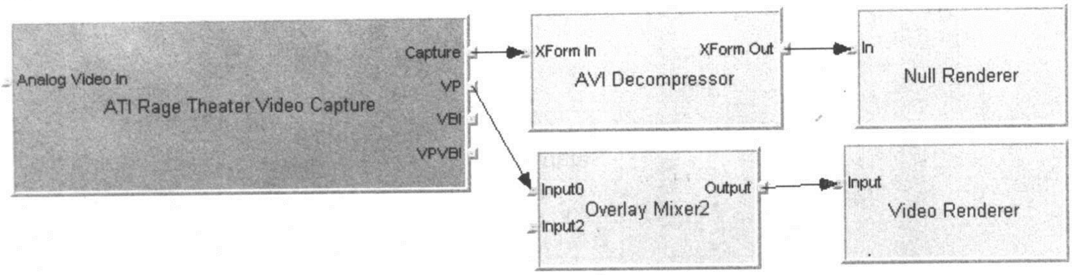

图 2.11　推模式 Filter Graph 参考图

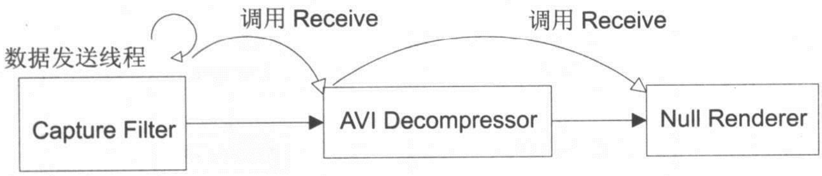

图 2.12　推模式数据传送流程

2.5.2　拉模式

拉模式最典型的情况发生在 File Source（文件源）中。这种源管理着数据，但它没有把数据“推”下去的能力，而要靠后面的 Filter 来“拉”。在如图 2.13 所示的情况下，Source Filter（图中显示为 aviDemo.AVI）的输出 Pin 上一定实现了一个 IAsyncReader 接口（当然此时 AVI Splitter 也没有必要在输入 Pin 上实现 IMemInputPin 接口了）。AVI Splitter 的输入 Pin 上一般会有一个“拉”数据的线程，不断调用 Source Filter 的输出 Pin 上的 IAsyncReader 接口方法来取得数据。在 AVI Splitter 内部，将从 Source Filter 中取得的数据进行分析、（音视频）分离，然后分别通过各个输出 Pin 发送出去。值得注意的是，AVI Splitter 的输出 Pin 往下的数据传送方式，与上述推模式是相同的，即通过调用下一级 Filter 的输入 Pin 上的 IMemInputPin::Receive 函数。如图 2.14 所示为这个传送过程的示意图。

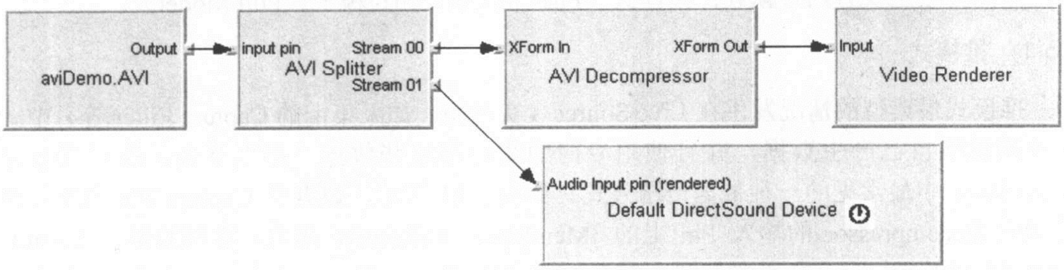

图 2.13　拉模式 Filter Graph 参考图

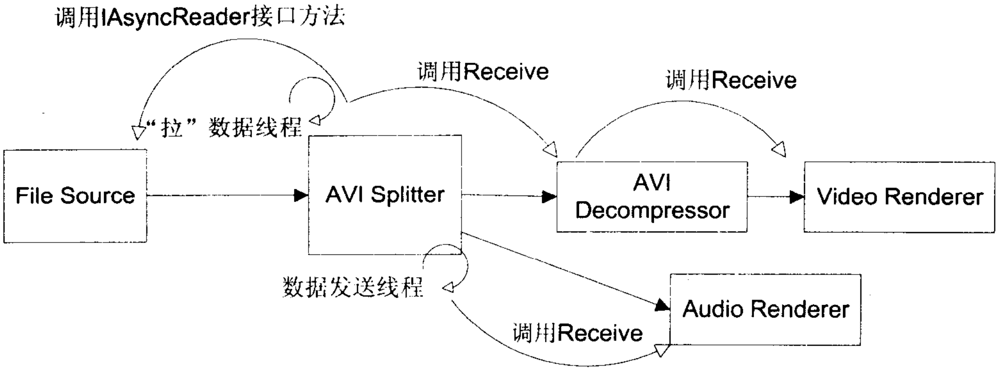

图 2.14　拉模式数据传送流程

提示

实际上，推模式和拉模式并不是对立的。如果把图 2.13 中的 Source Filter 和 AVI Splitter 看成另外一个“虚拟”的 Source Filter，那么这个虚拟的 Source Filter 传送数据的方式就完全变成推模式了。

从数据传送所使用的接口来看，Pin 之间可能以 IMemInputPin 连接，也可能以 IAsyncReader 连接。另外，还有一种连接方式，即通过 IOverlay 接口。传统的 Video Renderer 就支持这种方式（事实上，Video Renderer 的输入 Pin 上同时实现了 IMemInputPin 和 IOverlay 接口，不过这两个接口是不能同时使用的）。如图 2.11 的 Capture Filter （VP Pin） → Overlay Mixer2 → Video Renderer 这条链路中，Overlay Mixer2 到 Video Renderer 的连接使用的就是 IOverlay 接口。在这条链路中，Capture Filter 使用 DirectDraw Overlay 方式进行显示，所在在 Overlay Mixer2 与 Video Renderer 之间并没有实际的数据传输；Video Renderer 此时只相当于一个视频窗口的管理器。与 IOverlay 接口对应，Overlay Mixer2 在输出 Pin 上也实现了 IOverlayNotify 接口，用于接收视频窗口属性变更的通知。

提示

可以看出，DirectShow 总是使用专门的线程来传送数据。所以，也可以说 DirectShow 应用程序至少包含两条线程，一条应用程序主线程以及至少一条数据传送子线程。既然是多线程程序，就不可避免地会出现线程同步问题。将在后续章节重点讨论这个话题。

2.6　Filter 的状态转换

Filter 有 3 种状态：停止（Stopped）、暂停（Paused）和运行（Running）。其中，暂停可以理解为数据就绪状态，是为了快速切换到运行状态而设计的。作为一种中间状态，暂停也是停止与运行之间切换所必经的一种状态。

每个 Filter 上都实现了 IBaseFilter 接口，而 IBaseFilter 是从 IMediaFilter 继承来的。

Filter Graph Manager 正是通过 IMediaFilter 的接口方法来控制 Filter 的状态转换的。

提示　写 DirectShow 应用程序时，一般使用 Filter Graph Manager 上的 IMediaControl 接口来控制 Filter Graph 的状态转换。IMediaControl 有好几种接口方法，如 Run、Pause 和 Stop 等，分别用于实现 Filter Graph 的运行、暂停和停止。实际上，Filter Graph Manager 内部的实现，也是分别调用 Filter Graph 内所有 Filter 上的 IMediaFilter::Run、IMediaFilter::Pause 和 IMediaFilter::Stop 完成的。

对于大多数 Filter 来说（Source Filter 和 Transform Filter），暂停和运行两种状态的区别是可以忽略不计的，它们的任务仅仅是以最快的速度处理完数据，然后把数据往下传送。而对于 Rendering Filter（特别是 Video Renderer）来说，情况要复杂一点。

现在考虑如下一个构建完成的 Filter Graph：

Source Filter → Transform Filters → Rendering Filter

其中，Source Filter 是一个普通的工作在推模式下的 Filter（暂不讨论 Live Source 的情况）。当 Filter Graph 从停止状态转向暂停状态时，Source Filter 启动数据线程，并调用 Transform Filter 的输入 Pin 上的 IMemInputPin::Receive 方法，将数据以 Sample 形式往下传送。典型情况下，当 Transform Filter 接收到 Sample 并处理它之后，会在 Source Filter 的数据线程中将 Sample 继续往下传送给 Rendering Filter。当然，Transform Filter 也可以创建自己的工作者线程来继续完成数据的传送。此时，Rendering Filter 已经处于暂停状态，当它接收到第 1 个 Sample 的时候，就将数据线程阻塞住。如果是 Video Renderer，它还会将这个 Sample 帧显示在视频窗口中。

当 Filter Graph 从暂停状态转向运行状态时，Rendering Filter 只需简单地将数据线程脱离阻塞状态，就可以让整个 Filter Graph 继续运转起来了。Video Renderer 在这种情况下会首先显示在暂停状态下阻塞接收的 Sample 帧，然后每当接收到新的 Sample 时，将根据 Sample 上打的时间戳来安排显示。如果 Filter Graph 此时的参考时间还没有到当前接收到的 Sample 上的开始时间，则 Video Renderer 会将当前的 Sample 在输入 Pin 的 Receive 函数中（或者在另外的接收线程接收下一个新 Sample 时）阻塞住，直到到了当前 Sample 上的开始时间。

如果 Source Filter 是一个 Live Source（如视频采集卡的 Capture Filter 等），情况就有所不同了。因为这种源一般是通过实时采集产生数据的，暂停时阻塞了数据传送线程，在此之间采集的数据因为“过时”，自然也就失去了“实时”的意义。如果从暂停到运行中间等待了很长一段时间，那么把这段时间内采集的数据都缓存下来也是不可思议的。因此，在 DirectShow 设计中，Live Source 并不要求在进入暂停状态时就启动数据传送。Filter Graph Manager 调用 Filter 上的 IMediaFilter::GetState，如果返回 VFW_S_CANT_CUE，就认为该 Filter 已经成功转入暂停状态了。当然了，在这种情况下，Video Renderer 在暂停状态下是接收不到任何 Sample 的。

Filter Graph Manager 对 Filter 一个一个地实现状态转换的顺序很重要。处理不好，很容易造成 Sample 的丢失或者 Filter Graph 的死锁。因此，DirectShow 设计了一种从下往上的“回溯”方法。也就是说，Filter Graph Manager 总是从 Rendering Filter 开始执行状态转

换处理，沿着连接着的 Filter 链路，最后才处理到 Source Filter。最典型的状态转换发生在停止与暂停之间，下面具体介绍这两个过程。

➢ 停止→暂停

首先从 Rendering Filter 开始进行暂停状态的转换。这时，Filter 调用自己所有 Pin 上的 Active 函数进行初始化（一般 Pin 在 Active 函数调用中进行 Sample 内存的分配。如果是 Source filter，还将启动数据线程），使 Filter 处于一种就绪状态。Source Filter 是最后一个完成到就绪状态转换的 Filter。当 Source Filter 启动数据线程后，就开始往下发送 Sample；而当 Rendering Filter 接收到第 1 个 Sample 后就阻塞住，真正完成了状态转换。当 Filter Graph 中所有的 Rendering Filter 都完成了状态转换，即都接收到了执行暂停后的第 1 个 Sample（上文论述的 Live Source 情况除外），Filter Graph Manager 才认为状态转换完成。

➢ 暂停→停止

当 Filter 进入停止状态时，会调用自己所有 Pin 上的 Inactive 函数（一般 Pin 在 Inactive 函数调用中进行 Sample 内存的释放。如果是 Source Filter，还将终止数据线程）。各个 Filter 内部的处理还包括：释放所有阻塞住的 Sample，以使上一级 Filter 的 IMemAllocator::GetBuffer 调用脱离阻塞；终止所有在 Receive 函数中的等待，以使上一级 Filter 的 Receive 函数调用返回。另外，Filter 在停止状态下拒绝接收任何 Sample。就这样从 Rendering Filter 往上一级一级地脱离阻塞。当到达 Source Filter 的时候，可以确保数据线程的成功终止。

可以看到，Filter 在实现状态转换过程中，要处理一系列的函数调用。因为 DirectShow 应用程序肯定是多线程程序（一条应用程序主线程和至少一条数据传送子线程），在这一节的最后，我们还要来看一下 Filter 中的线程同步问题。

在 SDK 基类源代码 CBaseFilter 的定义中，用于标识 Filter 状态的成员变量的代码如下：

```
FILTER_STATE    m_State;
```

其中，Filter_State 是一个枚举类型，定义如下：

```
typedef enum _FilterState
{
    State_Stopped     = 0,                  // 停止状态
    State_Paused      = State_Stopped + 1,  // 暂停状态
    State_Running     = State_Paused + 1    // 运行状态
} FILTER_STATE;
```

还有如下用于同步 Filter 对象状态的同步对象指针的代码：

```
CCritSec        *m_pLock;
```

然而，在 CBaseFilter 上找不到用于同步数据流线程操作的同步对象。但可以从 CTransformFilter 类的定义中找到答案：

```
CCritSec m_csFilter;      // Filter 状态同步对象实例
CCritSec m_csReceive;     // 数据流线程同步对象实例
```

提示

CCritSec 是 SDK 提供的 CRITICAL_SECTION 的封装类，可用于线程同步。另外，SDK 还提供了一个类 CAutoLock，能够对 CCritSec 对象在一定的作用域内自动加锁、解锁，使用起来非常方便。代码如下：

```
 CCritSec csMyLock;  // 未锁定状态
 {
    CAutoLock cObjectLock(&csMyLock);  // 自动锁定
    // 执行受保护代码
} // 超出 cObjectLock 对象的作用域，cObjectLock 对象析构，自动解锁
```

为了防止资源使用冲突，必须使用上述同步对象使各个函数调用“串行化”。在应用程序主线程中调用的函数，必须使用 Filter 状态同步对象进行同步；在数据线程中调用的函数，必须使用数据流同步对象进行同步。两者不能混淆，否则很容易引起线程死锁。下面列出了 Filter 操作中大部分函数与调用者线程的关系。

应用程序主线程中调用的相关函数如下所示：

IBaseFilter::JoinFilterGraph
IMediaFilter::Pause
IMediaFilter::Run
IMediaFilter::Stop
IQualityControl::SetSink
IMediaSeeking::SetPositions
IMediaFilter::GetSyncSource
IMediaFilter::SetSyncSource
IBaseFilter::FindPin
IPin::Connect
IPin::ConnectedTo
IPin::ConnectionMediaType
IPin::Disconnect
IPin::ReceiveConnection
IMemInputPin::GetAllocator
IMemInputPin::NotifyAllocator
IPin::BeginFlush
IPin::EndFlush

数据线程中调用的相关函数：

IMemInputPin::Receive
IMemInputPin::ReceiveMultiple
IPin::EndOfStream
IMemAllocator::GetBuffer

两种线程都可能调用的函数：

IPin::NewSegment（这个函数的实现中不使用同步对象）

提示　可以参考 SDK 基类源代码中的 CBaseFilter 类、CTransformFilter 类和 CtransIn-PlaceFilter 类的实现，获得更多的关于各个函数中如何使用同步对象的认识。

2.7　媒体定位的实现

应用程序可以通过 Filter Graph Manager 上获得的 IMediaSeeking 接口，实现对流媒体的随机定位（Seeking）。实际上，IMediaSeeking 接口的真正实现在 Filter 上。

如图 2.15 所示，当应用程序通过 Filter Graph Manager 上申请获得 IMediaSeeking 接口执行定位操作（即调用 IMediaSeeking 的各接口方法）时，Filter Graph Manager 会首先从 Renderer Filter 上请求这个操作的执行。一般 Renderer Filter 不能真正执行定位操作，因此它将定位操作请求发给了上一级 Transform Filter 的输出 Pin（通过 Renderer Filter 的输入 Pin 可以获得 Transform Filter 的输出 Pin 的指针，然后在这个输出 Pin 上调用 QueryInterface 得到 IMediaSeeking 接口，再调用对应的接口方法）。Transform Filter 如果能够完成这个定位操作，则在这个 Filter 内执行后返回；如果不支持，则继续向上一级 Filter 的输出 Pin 请求，直到请求到 Source Filter。

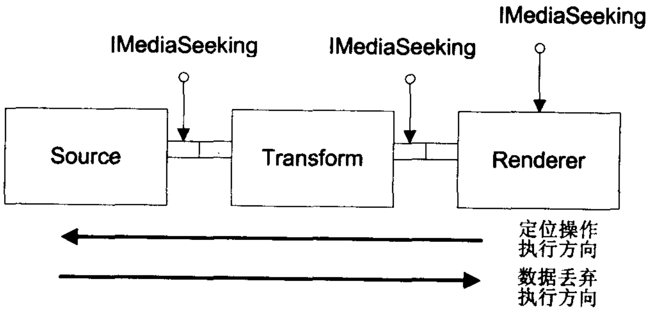

图 2.15　IMediaSeeking 接口的获取过程

实际上，一般在推模式的 Source Filter 的输出 Pin 上，或者拉模式下（Source 是标准的文件源）的 Parser Filter 或 Splitter Filter 的输出 Pin 上才可以真正执行定位操作。

对于 Filter 开发人员来说，如果写的是推模式 Source Filter，一般应该在 Filter 的输出 Pin 上实现 IMediaSeeking 接口；如果写的是 Transform Filter，只需要在输出 Pin 上将用户的定位操作请求向上传递给上一级 Filter 的输出 Pin；而如果写的是 Renderer Filter，就需要在 Filter 上（因为 Renderer Filter 没有输出 Pin）将用户的定位操作请求向上传递给上一级 Filter 的输出 Pin。另外，为了保证定位操作后媒体流的同步，如果真正实现 IMediaSeeking 各接口方法的 Filter 有多个输出 Pin，一般仅让其中一个 Pin 支持定位操作。对于定制的 Transform Filter，如果有多个输入 Pin，就需要自己决定当输出 Pin 接收到定位操作请求时，选择哪一条路径向上继续请求。

提示

可以借助 SDK 基类源代码的 CSourceSeeking、CPosPassThru、CRenderer-PosPassThru 等几个类（参见 ctlutil.cpp 文件）实现 Filter 上媒体的随机定位。另外，写 Transform Filter 时，如果 Filter 的父类采用 CTransformFilter 或 CtransIn-PlaceFilter，那么就不用考虑支持 IMediaSeeking 接口的问题了，因为基类中已经实现好了。代码如下：

```
STDMETHODIMP CTransformOutputPin::NonDelegatingQueryInterface(REFIID
riid, void **ppv)
{
    CheckPointer(ppv,E_POINTER);
    ValidateReadWritePtr(ppv,sizeof(PVOID));
    *ppv = NULL;
    if (riid == IID_IMediaPosition || riid == IID_IMediaSeeking) {
        ASSERT(m_pTransformFilter->m_pInput != NULL);
        if (m_pPosition == NULL) {
         // 创建能够将定位操作请求直接往上传递的 COM 组件
         HRESULT hr = CreatePosPassThru(
                  GetOwner(),
                  FALSE,
                  (IPin *)m_pTransformFilter->m_pInput,
                  &m_pPosition);
         if (FAILED(hr)) {
             return hr;
         }
        }
        return m_pPosition->QueryInterface(riid, ppv);
    } else {
       return  CBaseOutputPin::NonDelegatingQueryInterface(riid, ppv);
    }
}
```

应用程序能够在任何时候（无论 Filter Graph 是处于停止、暂停还是运行状态）对 Filter Graph 执行定位操作。但当 Filter graph 正在运行的时候，Filter Graph Manager 会先将 Filter Graph 暂停，执行完定位操作后，再重新运行。

IMediaSeeking 可以支持如下几种格式的定位操作：

```
TIME_FORMAT_FRAME              // 按视频帧定位
TIME_FORMAT_SAMPLE             // 按 Sample 计数定位
TIME_FORMAT_FIELD              // 按场定位
TIME_FORMAT_BYTE               // 按字节流中的偏移定位
TIME_FORMAT_MEDIA_TIME         // 按参考时间定位（以 100ns 为单位）
```

其中，TIME_FORMAT_MEDIA_TIME 是最基本的（也是默认的）格式。真正实现 IMedia-Seeking 接口方法的 Filter 也可以同时支持其他格式，但未必支持所有的格式。对于应用程序开发人员来说，当使用除 TIME_FORMAT_MEDIA_TIME 之外的其他格式时，最好先调用 IMediaSeeking::IsFormatSupported 进行确认。

IMediaSeeking 除了可以进行媒体的随机定位外，还可以用来调整媒体文件的回放速率，即调用接口方法 IMediaSeeking::SetRate，如参数为 2.0 表示为 2 倍速播放，参数为 0.5 表示以原来的一半速率播放。理论上，参数为负数可以实现“倒播”；但事实上，绝大部分 Filter 不支持这一特性。

当应用程序执行了 IMediaSeeking::SetRate 后，Filter Graph Manager 的内部响应如下：

（1）调用 IMediaSeeking::GetCurrentPosition 得到当前播放到的位置；

（2）如果当前 Filter Graph 处于暂停或运行状态，则调用 Stop；

（3）调用 IMediaSeeking::SetPositions 将步骤（1）中取得的当前位置重新设置一下；

（4）调用 IMediaSeeking::SetRate 设置新的回放速率；

（5）如果先前 Filter Graph 的状态为暂停或运行，则恢复这个状态。

相应地，各部分 Filter 上的响应如下：

- 推模式 Source Filter 或拉模式 Parser/Splitter Filter：这些 Filter 一般都对输出的 Sample 打时间戳，是真正响应 SetRate 的 Filter。当 Filter Graph 执行定位操作或者回放速率改变后，它们一般都会调用输出 Pin 上的 DeliverNewSegment 函数，提示一个新格式的流的开始；当然，此后输出的 Sample 的时间戳应该根据新的回放速率来计算。
- 解码 Filter：这种 Filter 一般不用对 SetRate 作出直接的响应，而是在接收到 NewSegment 调用后，把新的参数保存下来，并继续将 NewSegment 调用传递给下一级 Filter。
- Renderer Filter：Video Renderer 可以忽略回放速率的改变，因为最初发送 Sample 的 Filter 已经根据新的回放速率调整了时间戳。Video Renderer 仍然只须根据 Sample 上的时间戳来安排显示时机。但对于 Audio Renderer 来说，一般须要做一些相关的回放速率调整。

提示 用于媒体定位的还有另外一个接口 IMediaPosition。这是一个支持自动化（Automation）的接口，是为那些弱类型编程语言（如 VB 等）开发 DirectShow 应用程序而设计的。对于 Filter 开发人员来说，并不要求实现 IMediaPosition。Filter Graph Manager 会自动将 IMediaPosition 的接口方法调用转换为 IMediaSeeking 的接口方法调用。

2.8 质量控制的实现

数据线程是通过一种“压迫式”方式往下传送数据的。虽然 Video Renderer 接收到 Sample 后，会根据 Sample 上的时间戳来正确安排显示时机。但仅仅这样是不够的，它并不能改善 Filter Graph 运行时的性能。

因此，DirectShow 另外设计了一种“自适应”的反馈机制：质量控制（Quality Control）。运用这种机制，当 Video Renderer 发现数据线程发送数据太快时，就能发送一

个消息要求 Source Filter（拉模式下一般为 Parser Filter 或 Splitter Filter）减慢速度；当 Video Renderer 发现数据线程发送数据太慢时，就能发送一个消息要求 Source Filter（拉模式下一般为 Parser Filter 或 Splitter Filter）加快速度。

DirectShow 使用如下数据结构来描述这个质量控制消息：

```
typedef struct{
    QualityMessageType  Type;
    long                Proportion;
    REFERENCE_TIME      Late;
    REFERENCE_TIME      TimeStamp;
} Quality;
```

其中，Type 定义了消息的类型，取值为如下枚举类型：

```
typedef enum {
    Famine,       // 表示当前发送数据的速度太慢了
    Flood         // 表示当前发送数据的速度太快了
} QualityMessageType;
```

Proportion 以 1000 为基数定义了 Video Renderer“建议”的新的数据发送速率。例如，如果 Proportion 的值为 800，则建议新的发送速率为原速率的 80%；如果 Proportion 的值为 1200，则建议新的发送速率为原速率的 120%。

Late 定义了发送这个消息时，在 Video Renderer 上的 Sample 实际晚了多少时间（以 100ns 为单位）。

TimeStamp 定义了发送这个消息时，在 Video Renderer 上的 Sample 的时间戳的开始时间。

质量控制通过 IQualityControl 接口来实现。IQualityControl 接口方法见表 2.3。

表 2.3　IQualityControe 接口方法

接口方法	描述
Notify	发送质量控制消息
SetSink	设置一个自定义的质量控制处理器

质量控制在 Filter 上的具体实现如图 2.16 所示。

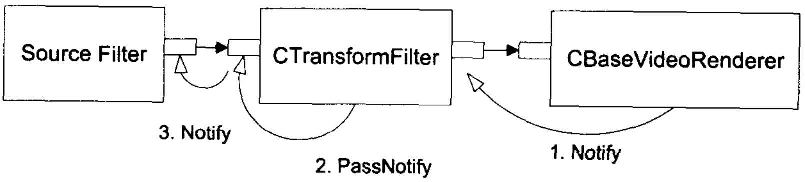

图 2.16　质量控制的处理过程

从图中可以看到，质量控制消息最初由 Video Renderer 发出，经过 Transform Filter 的输出 Pin，询问 Transform Filter 是否能够进行质量的改善；如果不能，则继续往上传递，直到 Source Filter 的输出 Pin 上。

在 SDK 基类的实现中还可以看到，CBasePin 上实现了 IQualityControl 接口，可以通

过 Pin 来实现质量控制消息的传递，而且默认的实现代码为：

```
STDMETHODIMP CBasePin::Notify(IBaseFilter * pSender, Quality q)
{
    UNREFERENCED_PARAMETER(q);
    UNREFERENCED_PARAMETER(pSender);
    DbgBreak("IQualityControl::Notify not over-ridden from CBasePin.
            (IGNORE is OK)");
    return E_NOTIMPL;
}
```

返回 E_NOTIMPL 表示没有实现真正的质量控制处理（留给派生类去实现）。而 CTransformFilter 的输出 Pin 实现代码如下：

```
STDMETHODIMP CTransformOutputPin::Notify(IBaseFilter * pSender, Quality q)
{
    UNREFERENCED_PARAMETER(pSender);
    ValidateReadPtr(pSender,sizeof(IBaseFilter));
    // 调用 Filter 上的 AlterQuality 函数，看是否能够执行质量控制
    // 返回 S_FALSE 表示该 Filter 不能处理质量控制
    HRESULT hr = m_pTransformFilter->AlterQuality(q);
    if (hr!=S_FALSE) {
        return hr;          // either S_OK or a failure
    }
    // S_FALSE means we pass the message on.
    // Find the quality sink for our input pin and send it there
    ASSERT(m_pTransformFilter->m_pInput != NULL);
    // 将质量控制消息往上一级 Filter 传递
    return m_pTransformFilter->m_pInput->PassNotify(q);
}
```

可以看到，当 Transform Filter 接收到质量控制消息后，首先调用 Filter 上的函数 AlterQuality，询问 Filter 是否可以进行质量的改善（默认实现中，AlterQuality 返回 S_FALSE 表示不进行处理），如果不可以，则调用输入 Pin 上的 PassNotify 函数，将质量控制消息传递给上一级 Filter 的输出 Pin。

这是 DirectShow 中一般的质量控制处理过程。另外，应用程序还可以实现自己的质量控制处理器，然后通过调用 IQualityControl::SetSink 方法设置给 Video Renderer。当然，这个自定义的质量控制处理器必须实现 IQualityControl 接口，并且通常是 Filter Graph 中的某一个组件对象。设置成功后，上述处理过程就改变了，质量消息会直接发送给自定义的处理器。但是，值得注意的是，大多数应用程序并不这么做，因为具体实现时还要考虑很多细节，并不像想象中那么简单。

提示　具体质量控制策略的实现取决于具体的 Filter 实现，可能是调整发送速度，也可能是丢失部分数据。所以，在实际应用中还要慎重使用质量控制机制，特别是对那些对数据有严格要求的应用。

2.9 音视频同步解决方案

在任何系统设计中，最关键的都是系统架构以及控制部分。DirectShow 系统也不例外。这个重要的部分就是 Filter Graph Manager，它向下直接控制 Filter Graph 中的所有 Filter，向上对应用程序提供编程接口。可惜的是，微软公司并没有提供这些功能控制的源代码。

本节介绍 Filter Graph Manager 实现的一个很重要的功能，即同步音视频的处理。简单地说，DirectShow 的解决方案就是为 Filter Graph 选择一个公共的参考时钟（Reference Clock），并且要求传送到 Renderer Filter 的每个 Sample 都打上时间戳（Time Stamp）；Video Renderer 或 Audio Renderer 根据 Sample 的时间戳以及参考时钟当前的参考时间，来正确安排 Sample 的播放时机。如果到达 Renderer Filter 的 Sample 晚了，则马上播放这个 Sample，并发送消息要求加快 Sample 的传送；如果 Sample 到达的时间早了，则 Renderer Filter 等待，直到 Sample 时间戳的开始时间再开始播放，并发送消息要求减慢 Sample 的发送。这里的消息控制过程也就是 2.8 节介绍的质量控制过程。

DirectShow 中规定，参考时钟以 100ns 作为计时精度，任何实现了 IReferenceClock 接口的对象都能担当参考时钟的角色。实际应用中，参考时钟往往是由某个 Filter 来实现的，如 Audio Renderer（因为声卡上一般都带有硬件定时器）；也可能是一个普通 COM 组件对象，如使用系统时钟时，可以创建 DirectShow 实现的一个 CLSID 为 CLSID_SystemClock 的组件。

一般每一个 Filter Graph 都必须使用一个参考时钟。Filter Graph Manager 有自己的一套选择参考时钟的策略：

（1）如果应用程序设置了一个参考时钟，则直接使用这个参考时钟；

（2）如果 Filter Graph 中有支持 IReferenceClock 接口的 Live Source，则选择这个 Live Source 作为参考时钟；

（3）如果 Filter Graph 中没有 Live Source，则从 Renderer Filter 开始依次往上选择一个实现 IReferenceClock 接口的 Filter。如果连接着的 Filter 都不能提供参考时钟，则再从没有连接的 Filter 中选择。还有一个优先情况，就是如果 Filter Graph 中含有一条音频播放的链路，则直接选择 Audio Renderer 这个 Filter（原因上面已经提及）；

（4）如果以上方法都找不到一个适合的 Filter，则选取系统参考时钟，即创建一个 CLSID_SystemClock 的组件对象实例。

提示 *应用程序给 Filter Graph 设置参考时钟的方法为：从 Filter Graph Manager 上获得 IMediaFilter 接口，并调用接口方法 IMediaFilter::SetSyncSource，参数为 IReferenceClock 接口对象的指针。如果这个指针参数为 NULL，表示 Filter Graph 不使用参考时钟，以最快的速度处理 Sample。另外，调用 IFilterGraph::SetDefaultSyncSource 可以恢复 Filter Graph Manager 默认的参考时钟。*

DirectShow 定义了两种时间：参考时间（Reference Time）和流时间（Stream Time）。

前者是从参考时钟返回的绝对时间（调用 IReferenceClock::GetTime），数值本身的意义取决于参考时钟的内部实现，利用价值不大；后者是两次从参考时钟读取的数值的差，实际应用于 Filter Graph 内部的同步。流时间的取值情况如下：

（1）Filter Graph 运行时，取值为当前参考时钟时间减去 Filter Graph 启动时的时间（启动时间是通过调用 Filter 上的 IMediaFilter::Run 设置的）；

（2）Filter Graph 暂停时，为暂停时刻的流时间；

（3）执行完一次随机定位（Seeking）操作后，数值复位至 0；

（4）Filter Graph 停止时，取值不确定。

提示　Sample 上的时间戳用的时间类型实际上就是流时间。虽然定义时间的数据类型名字也是 REFERENCE_TIME，但与参考时间是两码事。

通过查看 IMediaSample 的接口说明，可以发现 Sample 上有两种设置时间的方法：SetTime 和 SetMediaTime。通常所说的时间戳是指前者，Renderer Filter 也正是根据这个函数设置的时间来控制播放的；而后者对于 Filter 来说不是必需的，这要取决于具体 Filter 的实现，如通过 SetMediaTime 可以给每个发送出去的 Sample 依次打上递增的序号，当后面的 Filter 接收时，就可以判断传输的过程中 Sample 是否丢失。

时间戳包括开始时间和结束时间。在 DirectShow 中，并不是所有 Sample 都要求打时间戳。对于一些流式的压缩数据（如 MPEG），时间戳是很难打的，而且意义也不是很大（不过，压缩数据经过解码从 Filter 出来之后，在到达 Renderer Filter 之前，一般都要打上时间戳了）。

一般由推模式下的 Source Filter 或拉模式下的 Parser Filter 给 Sample 打时间戳。下面介绍不同的应用场合，Sample 时间戳的不同打法。

- 文件回放：第 1 个 Sample 的时间戳从 0 开始打起，后面 Sample 的时间戳根据 Sample 有效数据的长度和回放速率来确定。
- 音视频采集：原则上采集到的每一个 Sample 的开始时间都打上采集时刻的流时间。其中，对于视频采集来说，视频帧从 Preview Pin 出来的 Sample 是个例外。因为如果按上述一般方法打时间戳的话，每个 Sample 通过 Filter 链路传输，最后到达 Video Renderer 的时候都将是迟到的；Video Renderer 通过质量控制机制反馈给视频采集的 Capture Filter，会导致丢帧，所以，从 Preview Pin 出来的 Sample 都不打时间戳。对于音频采集来说，需要注意的是，音频采集的 Capture Filter 与声卡驱动程序各自使用了不同的缓存，采集的数据是由驱动程序定时（一般每次间隔不超过 10ms）从它的缓存中复制到 Filter 的缓存中的，音频 Sample 上的时间戳实际上是这种数据复制时刻的流时间，并不是严格意义上的“采集时刻的流时间”。
- 流合成：取决于最终合成输出的数据格式，可能要打时间戳，也可能不打。如 AVI 是一种恒定码流、不含时间戳的文件格式。DirectShow 提供的 AVI Mux Filter 内部就不用打时间戳。AVI Mux 假设输入的 Sample 都会以大致恒定的时间间隔到

达 Filter，如果前后 Sample 上的时间戳显示间隔很大（超过一帧的时间），AVI Mux 就会认为丢帧，于是写一个零数据长度的索引入口来表示丢掉的这一帧。在文件回放时，Sample 的时间戳是要运行时计算得到的（根据恒定的帧率）。

典型的 Live Source 有音视频采集卡、网络广播接收器等。一般 Filter 满足如下任何一个条件，Filter Graph Manager 就认为它是一个 Live Source：

（1）调用 Filter 上的接口方法 IAMFilterMiscFlags::GetMiscFlags，在返回值中包含 AM_FILTER_MISC_FLAGS_IS_SOURCE 标记，并且至少有一个输出 Pin 实现了 IAMPush-Source 接口；

（2）Filter 上实现了 IKsPropertySet 接口，并且有一个 Capture 输出 Pin（Pin 的类型为 PIN_CATEGORY_CAPTURE）。

Live Source 对音视频同步主要有延迟（Latency）和速率匹配（Rate Matching）两个方面的影响。

延迟是指 Filter 处理一个 Sample 所花费的时间。对于 Live Source 来说，它主要取决于使用缓存的大小，如采集 30fps 的视频，一般采集完一帧后才将数据以一个 Sample 发送出去，则这个 Filter 的延迟时间为 33ms；而音频一般缓存 500ms 后才发送一个 Sample，则它的延迟时间就为 500ms。这样的话，音频 Sample 与视频 Sample 到达各自的 Renderer Filter 的时间就会偏差 470ms，造成音频和视频的不同步。

默认情况下，Filter Graph Manager 是不会对这种情况进行调整的，除非应用程序调用它上面的接口方法 IAMGraphStreams::SyncUsingStreamOffset。此时，Filter Graph Manager 对所有实现 IAMPushSource 接口的 Filter 调用 IAMLatency::GetLatency 得到各个 Source 的延迟值（IAMPushSource 接口是从 IAMLatency 中继承来的），记下所有延迟值中的最大值，然后调用 IAMPushSource::SetStreamOffset 对各个 Source 设置偏移值。这样，在 Source Filter 产生 Sample 时，打的时间戳就会加上这个偏移量。

值得注意的是，通常 Preview Pin 出来的 Sample 不打时间戳，所以上述使用“偏移量”调整延迟时间的方法仅适合于通过 Capture Pin 进行预览的情况。

速率匹配主要是由于 Renderer Filter 和 Source Filter 使用不同的参考时钟而引入的。在这种情况下，Renderer Filter 对数据的播放要么太快，要么太慢。另外，一般 Live Source 不能控制输出数据的速率，所以必须在 Renderer Filter 上进行播放速率的匹配。因为人的听觉敏感度要大于视觉敏感度，所以 DirectShow 目前只在 Audio Renderer 上实现了速率匹配。速率匹配的算法大致如下：

- 如果 Filter Graph 没有使用参考时钟，则不进行速率匹配（Sample 到达 Renderer Filter 后都是立即处理的）；
- 如果 Filter Graph 使用了参考时钟，Audio Renderer 检查上游的 Filter 是否含有 Live Source，如果没有，则也不进行速率匹配；
- 如果有 Live Source，并且在这个 Filter 的一个输出 Pin 上实现了 IAMPushSource 接口，则调用 IAMPushSource::GetPushSourceFlags 接口方法取得 Live Source 的描述。如果描述标记中有 AM_PUSHSOURCECAPS_INTERNAL_RM，表示 Live

Source 已经实现了速率匹配，Audio Renderer 不用重复实现；如果标记中有 AM_PUSHSOURCECAPS_NOT_LIVE，表示这个 Source 并不是真正的 Live Source，Audio Renderer 也不用进行速率匹配；如果标记中含有 AM_PUSHSOURCECAPS_PRIVATE_CLOCK，表示 Live Source 使用自己内部的时钟来产生时间戳，Audio Renderer 必须根据 Sample 上的时间戳进行速率匹配（如果 Sample 上不含时间戳，认为这个标记是无效的，速率匹配算法往下继续）；

- 如果上述 IAMPushSource::GetPushSourceFlags 返回的标记为 0，则 Audio Renderer 的行为最终取决于当前 Filter Graph 使用的参考时钟和 Sample 上是否有有效的时间戳。如果 Audio Renderer 没有被用作当前 Filter Graph 的参考时钟，并且 Sample 上有有效的时间戳，则 Audio Renderer 根据 Sample 上的时间戳进行速率匹配；如果 Sample 上没有时间戳，Audio Renderer 尝试通过输入的数据率进行播放速率的匹配；如果 Audio Renderer 被用作当前 Filter Graph 的参考时钟，同样也是通过输入的数据率进行播放速率的匹配。

不难发现，参考时钟以及 Sample 使用的时间戳对于保证音频和视频同步是至关重要的。而 Video Renderer 和 Audio Renderer 作为音频和视频同步的“最终执行者”，实际也做了很多工作。我们或许要开发其他各种类型的 Filter，但一般这两个 Filter 是不用再开发的。一是因为 Renderer Filter 本身的复杂性，二是因为 DirectShow 会对这两个 Filter 不断升级，集成 DirectX 中其他模块的最新技术（如 DirectSound、DirectDraw、Direct3D 等）。

2.10　对硬件的支持

大家知道，为了提高系统的稳定性，Windows 对硬件操作进行了隔离，应用程序一般不能直接访问硬件。DirectShow Filter 工作在用户模式（User mode，操作系统特权级别为 Ring 3），而硬件工作在内核模式（Kernel mode，操作系统特权级别为 Ring 0），那么它们之间怎么协同工作呢？

DirectShow 的解决方案是为这些硬件提供包装（Wrapper）Filter，这种 Filter 能够工作在用户模式下，其外观和控制方法与普通 Filter 一样，而包装 Filter 在内部完成与硬件功能的交互。这样的设计可以使开发人员从支持硬件的特殊处理中解脱出来。

DirectShow 已经集成了很多包装 Filter，其中包括 Audio Capture Filter（实现在 qcap.dll 中）、VFW Capture Filter（实现在 qcap.dll 中，Filter 的 CLSID 为 CLSID_VfwCapture）、TV Tuner Filter（KsTune.ax，Filter 的 CLSID 为 CLSID_CTVTunerFilter）、Analog Video Crossbar Filter（KsXbar.ax）、TV Audio Filter（Filter 的 CLSID 为 CLSID_TVAudioFilter），以及为采用 WDM（Windows Driver Model）驱动程序的硬件而设计的 KsProxy Filter（KsProxy.ax）。如图 2.17 所示是各个包装 Filter 与硬件交互的结构图。可以看出，KsTune.ax、KsXbar.ax 这些包装 Filter 跟其他普通的 DirectShow Filter 处于同一个级别，可以协同工作；用户模式下的 Filter 通过 Stream class 控制硬件的驱动程序 minidriver。Stream class 和 minidriver 一起向上层提供系统底层级别的服务。

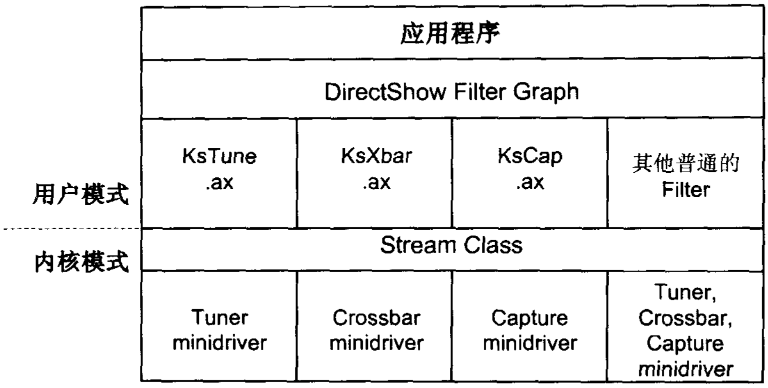

图 2.17 硬件的包装 Filter

值得注意的是，上述 Stream class 是一种驱动模型，它负责调用硬件的 minidriver；另外，Stream class 的功能还在于协调 minidriver 之间的工作，使得一些数据可以直接在内核模式下从一个硬件传输到另一个硬件（或同一个硬件上的不同功能模块），以提高系统的工作效率。

下面分别介绍几种常见硬件的情况。

- VFW 视频采集卡：这类硬件在市场上已经处于一种淘汰的趋势；新生产的视频采集卡一般采用 WDM 驱动模型。但是，DirectShow 为了保持向后兼容，还是专门提供了一个包装 Filter（VFW Capture Filter）支持这种硬件。
- 音频采集卡（声卡）：声卡的采集功能也是通过包装 Filter 来实现的，而且大部分都有混音功能。这个 Filter 一般有几个输入 Pin，每个 Pin 都代表一个输入，如 Line In、Microphone、CD、MIDI 等。值得注意的是，这些 Pin 代表的是声卡上的物理输入端子，在 Filter Graph 中是不用连接到其他 Filter 上的。

  声卡的输出可以有两个 Filter 供选择：DirectSound Renderer Filter 和 Audio Renderer (WaveOut) Filter。值得注意的是，这两个 Filter 不是上述意义上的包装 Filter，它们能够同硬件交互，是因为它们使用了 API 函数：前者使用了 DirectSound API，后者使用了 WaveOut API（并且后者输出音频的时候不支持混音）。

提示

其实，Video Renderer 能够访问显卡也是因为使用了 GDI、DirectDraw 或 Direct3D 等 API 函数，而不是包装 Filter。

- WDM 驱动的硬件（包括视频捕捉卡、硬件解压卡等）：这类硬件比 VFW 卡提供的功能更强大。就采集卡来说，WDM 比 VFW 提供更多的特性，包括采集格式的枚举、程序中的直接调节视频参数（如亮度、对比度、色调、饱和度等）、程序中直接选择输入端子、支持电视接收功能等。

上面已经介绍，DirectShow 使用 KsProxy.ax 这个包装 Filter 来支持 WDM 硬件。KsProxy.ax 实现了很多功能，所以有“瑞士军刀”的美誉，它还被称作为“变色龙”，因为该 Filter 上定义了统一的接口，而接口的实现因具体的硬件驱动程序而异。在 Filter

Graph 中，KsProxy Filter 显示的名字并不是 KsProxy，而是具体硬件的 Friendly Name（一般通过查找注册表得到）。可以通过 GraphEdit 程序，在 WDM Streaming 开头的目录中找到本机系统中安装的所有 WDM 硬件。

> 提示
> 在程序里可以通过系统枚举器枚举系统中安装的所有硬件。创建这些代表硬件的 Filter，同样也要通过系统枚举。枚举和创建的实现细节将在后续章节中介绍。

因为 KsProxy.ax 能够代表各种 WDM 的音视频设备，所以这个包装 Filter 的工作流程有点复杂。这个 Filter 预先不知道要代表哪种类型的设备，它必须首先访问驱动程序的属性集，然后动态配置 Filter 上应该实现的接口。当 KsProxy Filter 上的接口方法被应用程序或其他 Filter 调用时，它会将调用方法以及参数传递给驱动程序，由驱动程序最终完成指定功能。

除此以外，WDM 硬件还支持内核流（Kernel Streaming），即内核模式下的数据传输，而无须经过到用户模式的转换。因为内核模式与用户模式之间的相互转换需要花费很大的计算量。如果使用内核流，不仅避免了大量的计算，还避免了内核数据与主机内存之间的复制过程。在这种情况下，即使用户模式的 Filter Graph 中的 Pin 之间是连接的，也不会有实际的数据流动。典型的情况如带有 Video Port Pin 的视频采集卡，Preview 时显示的图像就是在内核模式下直接传送到显卡的显存中的。所以，在 Video Port Pin 后面截获数据流的努力是徒劳的。

总之，DirectShow 系统对硬件提供了充分的支持，这也是它强大功能的一个重要组成部分。其实，对于程序开发人员来说，上述 DirectShow 对硬件支持的原理，只需作为背景知识了解一下就行了，最主要还是应用，如如何创建硬件包装 Filter，如何设置这些 Filter 上的参数，如何实现音视频的预览、采集等。这些内容将在第 2 部分重点介绍。

2.11 VMR-9 的发布

VMR（Video Mixing Renderer）是 DirectShow 的新一代 Video Renderer。VMR 有两个版本：VMR-7 和 VMR-9。前者采用了 DirectDraw 7 技术，仅仅在 Windows XP 操作系统上可以获得，并且是 XP 上默认的用于视频显示的 Renderer（替代了传统的 Video Renderer）；后者采用了 Direct3D 9 的技术，是随 DirectX 9.0 一起发布的（任何安装了 DirectX 9.0 的操作系统上都能使用这个 Filter），但任何时候都不是默认的 Renderer。

> 提示
> VMR-7 在 Windows XP 上注册的 Filter 名称也为 Video Renderer，与传统的 Video Renderer 重名，但两者的 CLSID 是不同的，Merit 值也不一样。VMR-9 的注册名为 Video Mixing Renderer 9，Merit 值为 0x00200000（MERIT_DO_NOT_USE）。

与传统的 Video Renderer 相比，VMR 提供了更强劲的功能。

2.11.1 VMR 的新特性

VMR-7 支持如下新功能：

- 使用 Direct3D 硬件的 Alpha 混合能力，真正实现了多路视频流的合成（使用传统的 Video Renderer 实现多路视频合成，需要与 Overlay Mixer Filter 配合）；
- 支持在多路视频流输入到 VMR 之前，插入第三方的合成组件，用于实现效果（Effect）和过渡（Transition）；
- 实现了真正的无窗口模式显示（传统的 Video Renderer 在显示时总是创建一个视频窗口，应用程序如果要将视频内容显示在指定的 UI 窗口中，须要修改视频窗口的风格，并将它指定为 UI 窗口的子窗口）；
- 支持不显示（Renderless）回放模式。应用程序可以提供自己的内存分配器，以在视频显示之前优先获得图像数据；
- 更好地支持配置了多个显示器的 PC 机；
- 支持微软公司的 DirectX 视频加速架构；
- 支持在多个窗口中同时进行高质量的视频回放；
- 支持 DirectDraw 的独占模式（Exclusive Mode）；
- 完全向后兼容；
- 支持单帧播放以及可靠的当前帧捕捉方法；
- 为应用程序提供方便的静态图片（如一些图标、界面元素等）叠加功能。

VMR-9 除了支持上述所有的特性外，还具有如下功能：

- 支持用 Direct3D API 直接处理视频数据；
- 更好地支持隔行扫描的（Interlaced）视频内容；
- 支持在所有支持 DirectX 的操作系统上使用。

实际上，VMR 主要利用了显卡专有的图形处理能力（VMR 做视频的合成和显示时并不占用系统的 CPU 资源），. 它能够表现的性能对于硬件的依赖性很高。因此，当使用 VMR 进行视频处理的系统时，显卡最好满足如下条件：

- 支持 YUV 格式；
- 支持数据在 DirectDraw 表面从 YUV 到 RGB 的拉伸传送；
- 如果要支持多路视频合成，则至少要 16M 显存（具体需要的显存数量取决于输入图像的大小、当前的显示模式等）；
- 支持 RGB 的 Overlay 表面，或者提供与 YUV 的 Overlay 表面混合的能力；
- 支持视频解码的硬件加速；
- 高速的像素填充能力。

提示

VMR 要求显示器至少设置为 16 位色。有些显卡在设置了 24 位色的情况下不能执行 Direct3D 操作。

2.11.2 VMR 的结构

为了适应各种不同的应用，VMR 采用了模块化的设计策略。除了几个输入 Pin 外，VMR 还包含 2 至 5 个子组件（Sub-component），如图 2.18 所示。

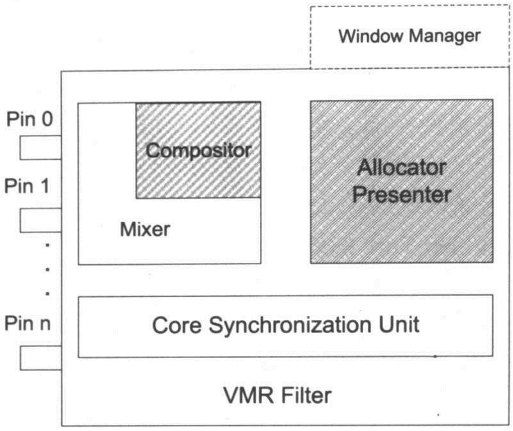

图 2.18　VMR 结构图

- 合成器（Mixer）：这是一个 COM 组件，而且仅当有多个视频流输入时被 VMR 装载。它用于收集各个输入流的信息，安排它们的 Z 坐标，决定何时从各个输入 Pin 上取得 Sample，并调用适当的图像合成器进行实际的图像合成。它同时还会为合成后的输出图像计算时间戳。如果应用程序中设置了要叠加静态图片的功能，合成器负责将这个图片叠加在最上面（即使其他输入流改变了 Z 坐标）。
- 图像合成器（Image Compositor）：这也是一个 COM 组件，而且真正地完成了图像合成功能。它将多路输入流合成到一个 DirectDraw 或 Direct3D 表面（这个表面由下述“内存分配管理器”提供）。VMR 内部使用了一个默认的图像合成器，以完成一般的二维图像 Alpha 混合。应用程序也可以提供自定义的图像合成器，以提供更多二维或三维的图像效果。
- 内存分配管理器（Allocator Presenter）：这也是一个 COM 组件，负责分配 DirectDraw 或 Direct3D 对象，并且与显卡通信。应用程序可以提供自己的内存分配管理器，用于创建和控制 DirectDraw 或 Direct3D 对象，并且在显示的时候能够获得访问视频图像数据的能力。
- 内核同步单元（Core Synchronization Unit）：这也是一个 COM 组件，用于安排视频帧的显示时机。它使用了 Filter Graph Manager 设置进来的参考时钟，并且提供了质量管理和帧报告功能。
- 窗口管理器（Window Manager）：这是一个静态库，而且仅在 VMR 工作在窗口模式下被使用。知道传统的 Video Renderer 实现了 IVideoWindow 和 IBasicVideo，为了保持向后兼容性，VMR 也实现了这两个接口。

从上述各个模块完成的功能看，不管 VMR 工作在何种应用环境中，内存分配管理器和内核同步单元总是必需的，根据不同的应用，其他的 3 个功能模块是可选的。

2.11.3　VMR 使用策略

安装了 DirectX 9.0 以后，就有 4 个 Video Renderer 可供选择使用：传统的 Video Renderer、Overlay Mixer、VMR-7 和 VMR-9。但是，没有一种 Renderer 是普遍适用的。

下面对选择合适的 Renderer 给出一些参考建议。

- 应用程序只是做基本的视频显示，不做视频流合成：如果想要更好地显示隔行扫描视频，或者需要调整视频的显示参数（如亮度、对比度、色调、饱和度等），那么可以选择 VMR-9；如果预先知道系统的硬件配置达不到 VMR-7 或者 VMR-9 的要求，那么应该选择传统的 Video Renderer。
- 应用程序基于使用 Video Port 的硬件（即视频数据在内核模式下直接传送到显卡的显存）：如果要求支持 Windows XP 以前版本的操作系统，则使用 Overlay Mixer（配合使用传统的 Video Renderer）；如果仅须支持 Windows XP，则使用 VMR-7（配合使用 Video Port Manager Filter）。注意：VMR-9 不支持 Video Port 的使用！
- 应用程序需要用到 VMR 特殊能力的情况：VMR-7 与 VMR-9 相比，优先使用 VMR-9，因为 VMR-9 采用了最新的 Direct3D 技术，而且能够在支持 DirectX 的所有操作系统上使用。如果我们的应用程序显式地指明了使用 VMR-7，那么这个程序只能运行在 Windows XP 上。

第 3 章　DirectX 媒体对象（DMO）

3.1　DMO 概 述

DirectX 媒体对象（DirectX Media Objects，DMO）是微软公司提供的另一种流数据处理解决方案。（DirectX 9.0 之前，DMO 是 DirectShow 的一部分。现在，DMO 已经分离出来，成为了 DirectX 家族中的一个独立分支。）

实际上，DMO 是一种实现了特定接口的 COM 组件（每个 DMO 上都实现了 IMediaObject 接口）。它跟 Filter 有很多相似之处，却比 Filter 更简单，使用起来更灵活。与 Filter 必须加入到 Filter Graph 中，并且完成连接后才能使用相比，DMO 的灵活性主要体现在如下 3 个方面：

（1）应用程序可以直接使用 DMO；

（2）在 DirectShow 中可以用 DMO Wrapper Filter 包装具体的 DMO 参与到 Filter Graph 中。此时，DMO 与普通 Filter 的区别对于应用程序是“透明的”；

（3）基于 DirectSound 的应用程序也可以使用各种声音效果的 DMO。

下面将 DMO 与 Filter 进行比较，以此来加深对 DMO 的认识。

DMO 与 Filter 的相似之处有以下几点：

（1）都是实现了特定接口的 COM 组件；

（2）都体现了一个输入到输出的过程。只不过在 Filter 中，输入输出通过 Pin 的概念来实现，Pin 也是 COM 组件；而 DMO 使用流（Stream）的概念，但流不是具体的对象实例，它仅仅通过索引号来区别；

（3）都有媒体类型的概念，用来描述实际处理的数据格式。不过，两者使用的数据结构不同，Filter 使用的是 AM_MEDIA_TYPE，而 DMO 使用的是 DMO_MEDIA_TYPE；

提示

DMO_MEDIA_TYPE 数据结构定义如下（实际上跟 AM_MEDIA_TYPE 相似）：

```
typedef struct _DMOMediaType {
    GUID      majortype;              // 主类型
    GUID      subtype;                // 辅助类型
    BOOL      bFixedSizeSamples;
    BOOL      bTemporalCompression;
    ULONG     lSampleSize;
    GUID      formattype;             // 格式细节类型
    IUnknown  *pUnk;
    ULONG     cbFormat;
    [size_is(cbFormat)] BYTE * pbFormat;
} DMO_MEDIA_TYPE;
```

（4）DMO 与 Transform Filter 相同，输入到输出的转换方式都有“复制”和“就地”

（In-place）处理两种。

DMO 与 Filter 的不同之处在于：

（1）Filter 之间需要连接，并协商媒体类型，而 DMO 不需要；
（2）Filter 必须自己分配 Sample 传送使用的内存，而 DMO 不需要；
（3）Filter 在处理完数据后，须要负责将这个数据往下“推”，而 DMO 不需要；
（4）Filter 须要与 Filter Graph Manager 进行事件交互，而 DMO 不需要；
（5）Filter 必须处理多线程同步问题，而 DMO 不需要。

上述 Filter 所做的工作（或者说 DMO 不做的工作）实际上都是由 DMO 的直接使用者来完成的。DMO 使用者分配数据内存，填入数据后送给 DMO 处理，等 DMO 处理完后再取出处理过的数据。整个过程中，DMO 完成的功能非常单一！

总之，使用 DMO 可以带来很多的好处：

（1）DMO 比 Filter 实现的功能要少很多，结构简单，使得 DMO“体积”很小；
（2）DMO 使用起来比 Filter 更灵活。DMO 的使用不需要 Filter Graph，应用程序可以直接与 DMO 交互；
（3）DMO 总是同步处理数据，不像 Filter 有独立的数据传送线程，须要考虑多线程编程问题；
（4）与传统的 ACM、VCM 编解码管理器相比，用 DMO 开发的编解码器是基于 COM 的，更易于扩展；
（5）与 ACM、VCM 编解码器相比，DMO 还支持多个输入和多个输出。

所以，微软公司推荐 DMO 做为新一代编解码器、声音效果处理的开发模型。也推荐用 DMO 替换 DirectShow Transform Filter。

3.2 DMO 的使用

下面重点介绍 DMO 的两种使用方法。

3.2.1 在应用程序中使用 DMO

在应用程序中可以直接使用 DMO，这是 DMO 的一大优势。DMO 的使用过程大致为：应用程序将需要处理的数据传给 DMO，DMO 处理完成后产生输出数据，应用程序再取出 DMO 的输出数据。下面具体看看这个过程。

设置输入输出媒体类型

假设已经创建了如下一个 DMO 对象：

```
IMediaObject * pDMO = NULL;
CoCreateInstance(CLSID_MyDMO, NULL, CLSCTX_INPROC,
IID_IMediaObject,(void **) &pDMO);
```

在 DMO 处理数据之前，必须首先对各个输入输出流进行媒体类型的设置（有一种流

叫可选流（Optional Stream），可以不设置媒体类型）。

一般，DMO 对象的各个流会提供首选的（Preferred）的媒体类型。枚举取出了索引为 0 的输入流上所有的首选媒体类型，代码如下：

```
DMO_MEDIA_TYPE mt;
DWORD dwType = 0;
while (hr = pDMO->GetInputType(0, dwType, &mt), SUCCEEDED(hr))
{
    // Examine this media type (not shown)
    /* ... */
    // Free the format block
    MoFreeMediaType(&mt);
    ++dwType;
}
```

注意，通过 IMediaObject::GetInputType 得到的首选媒体类型仅仅是一个参考，除此之外，还应注意以下几点：

（1）取出的首选媒体类型可能不包含完整的格式块，即 pbFormat 可能是无效的，但设置媒体类型的时候，必须提供完整的格式块；

（2）如果 DMO 有多个流，DMO 并不一定支持每个流上取得的首选媒体类型的各种组合；

（3）如果已经设置了一个流的媒体类型，可能会影响其他流上取出的首选媒体类型；

（4）并不是所有的流上都提供了首选媒体类型；

（5）在流上取得的首选媒体类型之外的媒体类型并不一定不被支持。

为一个流设置媒体类型的更通常的代码如下：

```
DMO_MEDIA_TYPE mt;
ZeroMemory(&mt, sizeof(DMO_MEDIA_TYPE));
// 为格式块分配内存
HRESULT hr = MoInitMediaType(&mt, sizeof(WAVEFORMATEX));
if (SUCCEEDED(hr))
{
    // 设置主类型、辅助类型、格式细节类型的 GUID
    mt.majortype  = MEDIATYPE_Audio;
    mt.subtype    = MEDIASUBTYPE_PCM;
    mt.formattype = FORMAT_WaveFormatEx;
    // 设置格式块参数
    WAVEFORMATEX *pWave = reinterpret_cast<WAVEFORMATEX*>(mt.pbFormat);
    pWave->wFormatTag = WAVE_FORMAT_PCM;
    pWave->nChannels = 2;
    pWave->nSamplesPerSec = 44100;
    pWave->wBitsPerSample = 16;
    pWave->nBlockAlign = (pWave->nChannels * pWave->wBitsPerSample) / 8;
    pWave->nAvgBytesPerSec = pWave->nSamplesPerSec * pWave->nBlockAlign;
    pWave->cbSize = 0;
    // 给流 0 设置媒体类型
```

```
    hr = pDMO->SetInputType(0, &mt, 0);
    // 释放格式块内存
    MoFreeMediaType(&mt);
}
```

当然，如果不想进行实际的媒体类型设置，仅仅是想让一个流检查一下是否支持某个媒体类型，可以使用 DMO_SET_TYPEF_TEST_ONLY 标记，代码如下（以 SetInputType 为例）：

```
if (S_OK == pDMO->SetInputType(0, &mt, DMO_SET_TYPEF_TEST_ONLY))
{
    // Media type is OK
}
```

返回 S_OK 表示支持，返回 S_FALSE 表示不支持。

另外，为 DMO 的一个流设置了媒体类型，可能会影响到其他流的设置，可以通过如下代码来清除一个流上的媒体类型（以 SetInputType 为例）：

```
pDMO->SetInputType(0, NULL, DMO_SET_TYPEF_CLEAR);
```

设置媒体类型的次序也有讲究。对于解码 DMO，一般先设置输入流上的媒体类型，然后在输出流上选择一种输出格式；而对于编码 DMO，一般先在输出流上设置媒体类型，然后在输入流上设置一种输入格式。

数据处理的过程

DMO 一般有两种数据处理方式：普通方式和“就地”方式。每个 DMO 对象都实现了 IMediaObject 接口，普通方式主要使用 IMediaObject::ProcessInput 和 IMediaObject::ProcessOutput 来处理数据的输入和输出；支持“就地”方式的 DMO 除了实现 IMediaObject 接口外，还实现了一个 IMediaObjectInPlace 接口，调用 ImediaObject-InPlace::Process 就可以实现数据的“就地”处理。

提示

支持“就地”处理的 DMO，切忌混合调用 IMediaObject 的 ProcessInput、ProcessOutput 接口方法和 IMediaObjectInPlace 的 Process 接口方法。

数据的“就地”处理方式比较简单，下面主要来讨论普通方式。在普通方式下，DMO 使用实现了 IMediaBuffer 接口的数据缓存对象，来完成数据的输入和输出（“就地”处理的 DMO 不必使用 IMediaBuffer 对象）。

首先要分配一定大小的数据缓存。通过调用 IMediaObject::GetInputSizeInfo 或 IMediaObject::GetOutputSizeInfo 可以分别得到输入和输出数据缓存的大小信息。这些信息包括缓存大小的最小需求（以 B 为单位）、是否要求数据对齐、输入流可能需要预留数据的数量等。

然后就是整个数据处理过程，如图 3.1 所示。应用程序首先将数据填入 IMediaBuffer 的缓存，调用 ProcessInput 传给 DMO（调用 IMediaObject::GetInputStatus 可以判断当前流是否可以输入更多的数据）。然后调用 ProcessOutput（该函数能够一次性操作所有输出

流），如果返回 S_OK，则成功取出一次数据，此时须要判断 ProcessOutput 函数调用的最后一个参数 pdwStatus 的输出，如果含有标记 DMO_OUTPUT_DATA_BUFFERF_INCOMPLETE，则说明还有数据在等待输出，须要再次调用 ProcessOutput；如果 ProcessOutput 返回 S_FALSE，表示没有数据输出，则跳回到数据输入流程；这样反复进行，直到所有数据输入完毕，最后在输入流上调用 IMediaObject::Discontinuity。

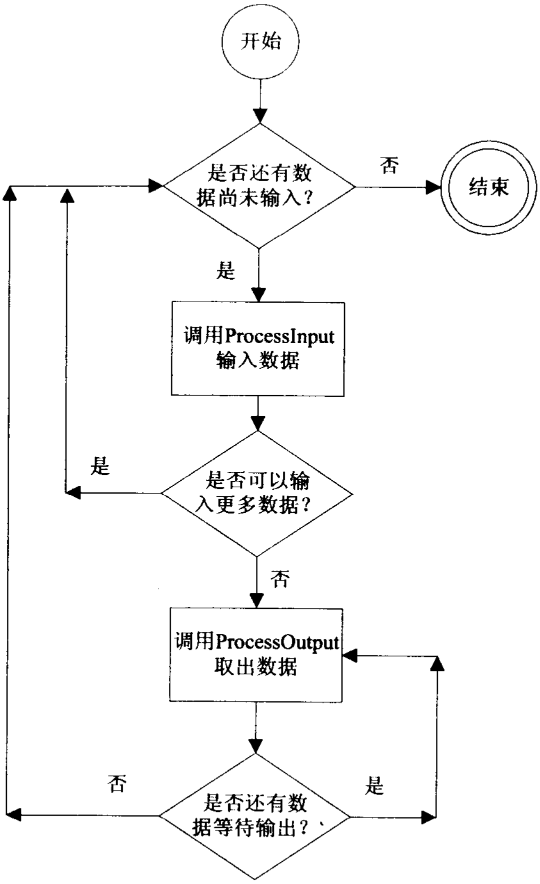

图 3.1　DMO 数据处理一般流程

提示　如果想要在整个处理过程中中断，可以调用 IMediaObject::Flush，这样 DMO 内部会释放所有占用的缓存资源。

数据缓存对象的实现

在 DMO 中，用于数据输入和输出的缓存是一个实现了 IMediaBuffer 接口的组件对象。应用程序必须负责实现它。

IMediaBuffer 有 3 个接口方法：GetBufferAndLength、GetMaxLength 和 SetLength。下面是这个组件对象的简单实现代码：

```
// CMediaBuffer class
#include <new>
#include <dmo.h>
class CMediaBuffer : public IMediaBuffer
{
private:
    DWORD        m_cbLength;       // 有效数据长度
    const DWORD  m_cbMaxLength;    // 缓存能够容纳的最大数据长度
    LONG         m_cRef;           // COM 引用计数器
    BYTE         *m_pbData;        // 数据缓存指针
public:
    CMediaBuffer(DWORD cbMaxLength) :
        m_cRef(0),
        m_cbMaxLength(cbMaxLength),
        m_cbLength(0),
        m_pbData(NULL)
    {
        m_pbData = new BYTE[cbMaxLength];
        if (!m_pbData) throw std::bad_alloc();
    }
    ~CMediaBuffer()
    {
        if (m_pbData) {
            delete [] m_pbData;
        }
    }
    // Function to create a new IMediaBuffer object and return
    // an AddRef'd interface pointer
    static HRESULT CreateBuffer(long cbMaxLen, REFIID iid, void **ppUnk)
    {
        try {
            CMediaBuffer *pBuffer = new CMediaBuffer(cbMaxLen);
            return pBuffer->QueryInterface(iid, ppUnk);
        }
        catch (std::bad_alloc)
        {
            return E_OUTOFMEMORY;
        }
    }
    // IUnknown methods
    STDMETHODIMP QueryInterface(REFIID riid, void **ppv)
    {
        if (ppv == NULL) {
            return E_POINTER;
        }
        if (riid == IID_IMediaBuffer || riid == IID_IUnknown) {
            *ppv = static_cast<IMediaBuffer *>(this);
            AddRef();
            return S_OK;
        }
```

```
            *ppv = NULL;
            return E_NOINTERFACE;
        }
        STDMETHODIMP_(ULONG) AddRef()
        {
            return InterlockedIncrement(&m_cRef);
        }
        STDMETHODIMP_(ULONG) Release()
        {
            LONG lRef = InterlockedDecrement(&m_cRef);
            if (lRef == 0) {
                delete this;
                // m_cRef is no longer valid! Return lRef
            }
            return lRef;
        }
        // IMediaBuffer methods
        STDMETHODIMP SetLength(DWORD cbLength)
        {
            if (cbLength > m_cbMaxLength) {
                return E_INVALIDARG;
            } else {
                m_cbLength = cbLength;
                return S_OK;
            }
        }
        STDMETHODIMP GetMaxLength(DWORD *pcbMaxLength)
        {
            if (pcbMaxLength == NULL) {
                return E_POINTER;
            }
            *pcbMaxLength = m_cbMaxLength;
            return S_OK;
        }
        STDMETHODIMP GetBufferAndLength(BYTE **ppbBuffer, DWORD *pcbLength)
        {
            if (ppbBuffer == NULL || pcbLength == NULL) {
                return E_POINTER;
            }
            *ppbBuffer = m_pbData;
            *pcbLength = m_cbLength;
            return S_OK;
        }
};
```

DMO 可选流的处理

DMO 的一些输出流可以被指定为可选流。可选流用于辅助说明主输出流，它能够产生输出数据，但是应用程序可以完全或者部分丢弃这些数据。

可以通过 IMediaObject::GetOutputStreamInfo 接口方法判断一个流是不是可选流，此

时参数 pdwFlags 一般返回 DMO_OUTPUT_STREAMF_DISCARDABLE 或 DMO_OUTPUT_STREAMF_OPTIONAL。这两个标记的区别是：如果是前者，则流上一定要设置媒体类型；而如果是后者，流上的媒体类型可以设置（用于部分丢弃数据的情况），也可以不设置（用于完全丢弃的情况）。

IMediaObject::ProcessOutput 函数的原型如下：

```
HRESULT ProcessOutput(
    DWORD dwFlags,
    DWORD cOutputBufferCount,
    DMO_OUTPUT_DATA_BUFFER* pOutputBuffers,
    DWORD* pdwStatus
);
```

当调用 ProcessOutput 在可选流上操作时，一般须要将这个流对应的 pOutputBuffers 的 pBuffer 成员指定为 NULL，并且将 dwFlags 指定为 DMO_PROCESS_OUTPUT_DISCARD_WHEN_NO_BUFFER。

对于任何一个流来说，调用 ProcessOutput 时上述 pBuffer 成员指定为 NULL，DMO 会尽量丢弃这个流上的数据。如果这个流是可选流，那么数据肯定能够丢弃；如果不是，那么数据并不一定能够丢弃。如果数据丢弃不成功，DMO 会设置标记 DMO_OUTPUT_DATA_BUFFERF_INCOMPLETE，则告诉调用者 DMO 中还有数据等待读出。而如果 pBuffer 成员已经指定为 NULL 了，但 dwFlags 没有指定为 DMO_PRO- CESS_OUTPUT_DISCARD_WHEN_NO_BUFFER，则 DMO 不丢弃数据，这种情况下，DMO 要么将数据在内存缓存着，要么直接返回一个错误码。

3.2.2 在 DirectShow 中使用 DMO

DirectShow 提供了一个 DMO Wrapper Filter，使得 DMO 能够很方便地参与到 Filter Graph 中。实际上，从 COM 的原理上说，DMO Wrapper Filter 聚合了 DMO 组件。这样从这个 Filter 上能够得到 DMO 上的所有接口。但是，应用程序千万不要这么做，因为所有使用 DMO 相关的操作都由 DMO Wrapper Filter 统一处理了。

在 Filter Graph 中使用一个 CLSID_MyDMO 的 DMO 的一般创建步骤如下：

（1）使用 CoCreateInstance 函数创建一个 DMO Wrapper Filter 实例；

（2）从创建成功的 DMO Wrapper Filter 上获得 IDMOWrapperFilter 接口；

（3）调用 IDMOWrapperFilter::Init，并指定 DMO 的 CLSID 和类型目录。

下面是上述步骤的代码实现（假设 CLSID_MyDMO 注册在 DMOCATEGORY_VIDEO_EFFECT 下）：

```
// 创建一个 DMO Wrapper Filter 实例
IBaseFilter *pFilter;
HRESULT hr = CoCreateInstance(CLSID_DMOWrapperFilter, NULL,
    CLSCTX_INPROC_SERVER, IID_IBaseFilter, reinterpret_cast<void**>
(&pFilter));
if (SUCCEEDED(hr))
{
```

```
        // 获得 IDMOWrapperFilter 接口
        IDMOWrapperFilter *pDmoWrapper;
        hr = pFilter->QueryInterface(IID_IDMOWrapperFilter, (void**)
        &pDmoWrapper);
        if (SUCCEEDED(hr))
        {
            // 初始化 DMO Wrapper Filter
            hr = pDmoWrapper->Init(CLSID_MyDMO, DMOCATEGORY_VIDEO_EFFECT);
            pDmoWrapper->Release();
            if (SUCCEEDED(hr))
            {
                // 将完成初始化的 DMO Wrapper Filter 加入到 Filter Graph 中
                hr = pGraph->AddFilter(pFilter, L"My DMO");
            }
        }
        pFilter->Release();
    }
```

除了上述直接的方法外，还可以使用系统枚举法创建 DMO。DMO 类型目录与 DirectShow 类型目录的对应关系见表 3.1。

表 3.1　DMO 类型目录与 DirectShow 类型目录的对应关系

DMO 目录	DirectShow 中的等价目录
DMOCATEGORY_AUDIO_ENCODER	CLSID_AudioCompressorCategory
DMOCATEGORY_AUDIO_DECODER	CLSID_LegacyAmFilterCategory
DMOCATEGORY_VIDEO_ENCODER	CLSID_VideoCompressorCategory
DMOCATEGORY_VIDEO_DECODER	CLSID_LegacyAmFilterCategory

提示　如何在 DirectShow 中枚举特定的类型目录并创建 Filter，将在后续的章节介绍。

在 DirectShow 中使用 DMO 有如下一些限制：

（1）DMO Wrapper Filter 不能支持没有输入流，或者没有输出流的 DMO，也不支持多个输入流的 DMO；

（2）DMO Wrapper Filter 的输入 Pin 只支持使用 IMemInputPin 接口传送数据；

（3）当前版本的 DES（DirectShow Editing Services）不支持 DMO 的效果（Effect）和过渡（Transition）。

3.3　DMO 的开发要点

DMO 是一种比较简单的 COM 组件，所以一般可以使用 VC 的 ATL（Active Template Library）项目来开发。DMO 组件开发的最低要求为：

（1）支持聚合（Aggregation）；

（2）实现 IMediaObject 接口；

（3）线程模型为 both，要求能够运行在自由线程环境；

（4）如果是音效 DMO，为了能在 DirectMusic 或 DirectSound 中使用，必须实现 IMediaObjectInPlace 接口。

另外，DMO 还可以实现如下一些可选接口：

- IspecifyPropertyPages 和 IPropertyPage：提供属性页让用户设置属性；
- IPersistStream：用于将 DMO 内部状态保存到永久介质（如文件）；
- IAMStreamConfig 和 IAMVideoCompression：如果开发的是编码 DMO，可以用这些接口设置输出的压缩格式。

3.3.1 DMO 中的媒体类型

媒体类型是用来描述流数据具体格式的，这一点跟 DirectShow Filter 是很相似的。不同的是，DMO 使用 DMO_MEDIA_TYPE 数据结构，而 Filter 使用 AM_MEDIA_TYPE 数据结构。

DMO 中还有另外一个数据结构 DMO_PARTIAL_MEDIATYPE，定义如下：

```
typedef struct _DMO_PARTIAL_MEDIATYPE {
    GUID type;       // 主类型
    GUID subtype;    // 辅助类型
} DMO_PARTIAL_MEDIATYPE, *PDMO_PARTIAL_MEDIATYPE;
```

DMO_PARTIAL_MEDIATYPE 同样可以用于描述媒体类型，只是它是 DMO_MEDIA_TYPE 的一个子集（仅仅包含了相当于 DMO_MEDIA_TYPE 的 majortype 和 subtype），是一个不完全的描述。这个数据结构多用于注册函数 DMORegister（将 DMO 支持的媒体类型写到注册表，以后可以使用 DMOEnum 指定某个媒体类型进行枚举得到相应的 DMO 组件）。

一般 DMO 的输入流和输出流上都可以提供首选媒体类型，通过 IMediaObject::GetInputType 或 IMediaObject::GetOutputType 可以得到它们，但这些媒体类型仅仅只能作为实际设置媒体类型时的参考（首选媒体类型可能并不完整，如不含格式数据块，但设置用的媒体类型必须是完整的）。流与流的媒体类型可能是相关的。一个流上是否能够接受某个媒体类型，有时候会受其他流的影响（这个流可能已经设置了一个媒体类型）。下面看看几种典型的应用情况。

视频解码 DMO

输入和输出一般都保持相同的帧率和相同的图像大小等属性。因此，这种 DMO 一般先在输入流上设置媒体类型，这个媒体类型还会“部分地”决定输出流的媒体类型。在开发这种 DMO 的时候，当输入流上没有设置媒体类型之前，通常的做法是不要在输出流上提供首选媒体类型，或者仅仅提供不完全的媒体类型（如不含具体格式数据块，而仅以 subtype 来表示该 DMO 支持的解码输出格式）。

音频解码 DMO

因为解码后的音频格式有限，所以这种 DMO 可以在输入流没有设置媒体类型之前，在输出流上提供完整的首选媒体类型集。

压缩 DMO

大多数情况下，视频压缩器在输入流没有设置媒体类型之前，输出流不能提供完整的首选媒体类型。不过，DMO 可以返回一个不带具体格式数据块的不完全媒体类型。对于压缩器（音频或视频）来说，一般先要在输出流上设置各种参数（如比特流大小等）。当输入流设置了一种媒体类型后，一般在输出流上应该至少提供一种完整的首选媒体类型。

3.3.2 DMO 的 ATL 实现

DirectX SDK 提供了一个模版类 IMediaObjectImpl，使得在 ATL 下开发 DMO 非常容易。这个模版类的声明如下：

```
template <class _DERIVED_, int NUMBEROFINPUTS, int NUMBEROFOUTPUTS>
class IMediaObjectImpl : public IMediaObject
```

其中，参数_DERIVED_表示 DMO 实现类名；NUMBEROFINPUTS 表示输入流的个数；NUMBEROFOUTPUTS 表示输出流的个数。现在假如要定义一个 DMO 实现类 CMyDMO，有一个输入流和两个输出流，则应该如下声明：

```
class CMyDMO : public IMediaObjectImpl< CMyDMO, 1, 2>
```

至于 IMediaObject 接口函数的实现，模版类里已经作了很多工作，包括如下一些方面：

- 基本的参数检查，如指针参数是否为 NULL，流的索引是否越界，标记是否有效等；
- 在 DMO 的接口方法中都使用了锁（调用函数 Lock 和 Unlock），使得接口函数调用“串行化”，保证了 DMO 是线程安全的（Thread-safe）；
- 保存应用程序设置的媒体类型，并提供这些媒体类型的访问接口；
- 在所有非可选流没有设置媒体类型之前，阻止数据处理流程的启动，并且保证在数据流程启动之前调用 IMediaObject::AllocateStreamingResources（分配 DMO 需要的资源）。

另外，模版类几乎为所有的 IMediaObject 接口方法都生成了对应的内部函数，函数名形式为 Internalxxx（如 GetInputType 对应的内部函数为 InternalGetInputType）。在具体实现的时候，只须要实现这些内部函数就行了。这样的设计，使得我们的开发工作可以更多地集中在 DMO 的功能实现上。

模版类 IMediaObjectImpl 并不直接支持 IMediaObjectInPlace 接口。如果想要实现“就地”处理，须要将 DMO 实现类从 IMediaObjectInPlace 中派生出来，代码如下：

```
class CMyDMO : public IMediaObjectImpl< CMyDMO, 1, 2>
             , public IMediaObjectInPlace
```

并实现 IMediaObjectInPlace 的所有接口方法。

提示 开发完成 DMO 组件后，可以使用 SDK 目录 Bin\DXUtils 下的 dmotest.exe 程序进行测试。如何使用这个程序，可以参考相同目录下的 dmotest.chm 帮助文档。

3.3.3 DMO 的注册

DMO 作为一种 COM 组件，使用前必须先注册。注册是通过 DLL 的导出函数 DllRegisterServer 来实现的，在 DllRegisterServer 中可以直接调用 API 函数 DMORegister。

DMO 组件可以注册到一个类型目录中，这样，API 函数 DMOEnum 就可以通过枚举这个目录来访问 DMO 组件。DMO 已经定义的目录见表 3.2（在头文件 Dmoreg.h 中声明）。

表 3.2 DMO 中已经定义的目录

目录的 GUID	描述
DMOCATEGORY_AUDIO_DECODER	音频解码器
DMOCATEGORY_AUDIO_EFFECT	音效
DMOCATEGORY_AUDIO_ENCODER	音频编码器
DMOCATEGORY_VIDEO_DECODER	视频解码器
DMOCATEGORY_VIDEO_EFFECT	视频效果
DMOCATEGORY_VIDEO_ENCODER	视频编码器
DMOCATEGORY_AUDIO_CAPTURE_EFFECT	音频采集效果

一个 DllRegisterServer 函数的实现代码如下，它注册了一个支持 PCM 音频的输入和输出音效的 DMO，类型目录为 DMOCATEGORY_AUDIO_EFFECT。

```
STDAPI DllRegisterServer(void)
{
    // Register the DMO as a PCM audio effect DMO
    DMO_PARTIAL_MEDIATYPE mt;
    mt.type    = MEDIATYPE_Audio;
    mt.subtype = MEDIASUBTYPE_PCM;
    HRESULT hr = DMORegister(
        L"MyDMO",                      // Friendly name
        CLSID_MyDMO,                   // CLSID
        DMOCATEGORY_AUDIO_EFFECT,      // Category
        0,                             // Flags
        1,                             // Number of input types
        &mt,                           // Array of input types
        1,                             // Number of output types
        &mt);                          // Array of output types

    if (FAILED(hr)) return hr;

    // 注册 COM 服务器，DMO 没有类型库支持
    return _Module.RegisterServer(FALSE);
}
```

类似地，解除 DMO 注册的实现代码如下：

```
STDAPI DllUnregisterServer(void)
{
    DMOUnregister(CLSID_MyDMO, DMOCATEGORY_AUDIO_EFFECT);
    return _Module.UnregisterServer(TRUE);
}
```

提示　当 DMO 在 Filter Graph 中应用时，DirectShow 为其分配了一个默认的 Merit 值。在开发 DMO 的时候，也可以定制这个 Merit 值，即在注册表中的 HKEY_CLASSES_ROOT\CLSID\ CLSID_MyDMO 子键下加一个 DWORD 类型的值，值的名字为 Merit。如果在 ATL 中实现，可以在.rgs 文件中编辑：

```
HKCR
{
   NoRemove CLSID
   {
           ForceRemove {df20ddfa-0d19-463a-ab46-e5d8ef6efd69} = s '
           DMO Sample Class'
           {
                   InprocServer32 = s '%MODULE%'
                   {
                           val ThreadingModel = s 'Both'
                   }
                   val MERIT = d '0'
           }
   }
}
```

第 2 部分　DirectShow 开发与应用

第 4 章　Filter 组件的开发

4.1　开发环境的配置

Filter 是一种进程内的 COM 组件。DirectShow SDK 提供了一套基类源代码，这些源代码可以大大降低开发 Filter 的难度。我们所要做的就是按以下要求配置好 Filter 的开发环境。

- 确认已经编译过基类源代码，生成一个静态库（一般 Debug 版本生成 Strmbasd.lib，Release 版本生成 Strmbase.lib）；
- 使用 VC 的向导生成一个普通的 Win32 动态链接库作为 Filter 项目，即在 VC 的菜单中选择 File | New | Projects，在弹出的对话框中选择 Win32 Dynamic-Link Library；
- 实现 Filter 的源文件中一般要包含 Streams.h 头文件；
- 一般使用_stdcall 函数调用协议，即在 VC 的菜单中选择 Project | Settings | C/C++，在弹出的对话框中的 Category 中选择 Code generation，然后在 Calling convention 中选择_stdcall；
- 使用多线程的 C 语言运行时库，即在 VC 的菜单中选择 Project | Settings | C/C++，在弹出的对话框中的 Category 中选择 Code generation，然后在 Use run-time library 中，Debug 版选择 Debug Multithreaded，Release 版选择 Multithreaded；
- 配置必要的链接库文件，即在 VC 的菜单中选择 Project | Settings | Link，在弹出的对话框中的 Category 中选择 General，然后在 Object/library modules 中输入如下代码：

```
Debug 版本                  Strmbasd.lib, Msvcrtd.lib, Winmm.lib
Release 版本                Strmbase.lib, Msvcrt.lib, Winmm.lib
```

  并且选中 Ignore all default libraries。
- 定义一个.def 文件用于导出 DLL 函数。假设 Filter 的文件名为 MyFilter.ax，.def 文件的一般格式如下：

```
LIBRARY MyFilter.ax
EXPORTS
DllMain                   PRIVATE
    DllGetClassObject     PRIVATE
    DllCanUnloadNow       PRIVATE
    DllRegisterServer     PRIVATE
    DllUnregisterServer PRIVATE
```

- 定义 DLL 的入口函数，代码如下：

```
extern "C" BOOL WINAPI DllEntryPoint(HINSTANCE, ULONG, LPVOID);
BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, LPVOID lpReserved)
{
    return DllEntryPoint((HINSTANCE)(hModule), dwReason, lpReserved);
}
```

提示　也可以不定义和导出 DllMain 函数，而在 VC 的项目设置中指定一个入口函数，比如在 VC 的菜单中选择 Project | Settings | Link，在弹出的对话框中的 Category 中选择 Output，然后在 Entry-point symbol 中输入 DllEntryPoint@12。

- 如果安装的 SDK 版本是 9.0 以前的，编译 Debug 版本时，还需确认定义了 DEBUG 宏，即在 VC 的菜单中选择 Project | Settings | C/C++，在弹出的对话框中的 Category 中选择 Preprocessor，然后在 Preprocessor definitions 中确认已经定义 DEBUG。

提示　配置 Filter 的开发环境还有其他“捷径”，如对照着 SDK 中的 Filter 例子项目逐项 Copy-Paste 过来；或者复制一个 Filter 例子项目，并直接在它的基础上修改，开发新的 Filter。

另外，除了配置 Filter 项目外，还需将 DirectX SDK 的 Include 和 Lib 目录配置到 VC 的系统目录中去，并且放在标准的 VC 目录之前，以保证编译器能够拿到最新版本的源文件。配置方法为，打开 VC，选择菜单项 Tools | Options | Directories，在弹出的对话框中的 Show directories for 中选择 Include files，配置如下（假设 DirectX SDK 安装在 D:\DXSDK90 目录下，VC 安装在 C:\Program Files 下）：

```
D:\DXSDK90\Include
D:\DXSDK90\SAMPLES\C++\DIRECTSHOW\BASECLASSES
D:\DXSDK90\SAMPLES\C++\COMMON\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\ATL\INCLUDE
```

再在 Show directories for 中选择 Library files，配置如下：

```
D:\DXSDK90\Lib
D:\DXSDK90\SAMPLES\C++\DIRECTSHOW\BASECLASSES\DEBUG
D:\DXSDK90\SAMPLES\C++\DIRECTSHOW\BASECLASSES\RELEASE
C:\PROGRAM FILES\MICROSOFT SDK\LIB
C:\Program Files\Microsoft Visual Studio\VC98\LIB
C:\Program Files\Microsoft Visual Studio\VC98\MFC\LIB
```

4.2　SDK 基类分析

DirectShow SDK 提供了一套开发 Filter 的基类源代码。虽然这些源代码对于开发 Filter 来说不是必需的，但我们还是强烈推荐使用。因为基于这些基类开发 Filter 将大大简化开发过程。下面重点介绍这些基类的设计结构。

4.2.1 CBaseObject

大部分 SDK 类都从 CBaseObject 类（参见 combase.cpp 的实现）中继承而来。这个类提供了 Debug 时的帮助功能，如在运行时刻跟踪 Filter 实现的 DLL 中活动对象的数量。当在 DLL 中还有活动对象存在而试图卸载时（一般是进行了不恰当的引用计数），CBaseObject 类能够提供报错功能。

4.2.2 CUnknown

作为 COM 组件，最基本的当然是 IUnknown 接口的实现。SDK 提供了 CUnknown 类（参见 combase.cpp 的实现），SDK 中实现了 COM 接口的类都是直接或间接从这个类继承来的。

如图 4.1 所示，CUnknown 类从 CBaseObject 中继承而来。另外，CUnknown 类还实现了 INonDelegatingUnknown 接口，用于支持引用计数、接口查询、COM 组件“聚合”等。

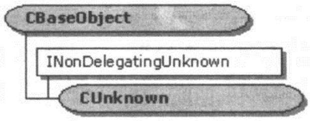

图 4.1　CUnknown 类结构

CUnknown 类的使用方法如下：

（1）从 CUnknown 派生一个子类，并在子类的 public 区加入 DECLARE_IUNKNOWN 宏；

（2）重写 NonDelegatingQueryInterface 函数，用以支持 IUnknown 外的其他接口；

（3）在子类的构造函数中调用 CUnknown 的构造函数。

4.2.3 CBaseFilter

最基本的 Filter 由 CBaseFilter 类（参见 amfilter.cpp）实现。

如图 4.2 所示，作为 Filter 的基本特征，CBaseFilter 实现了 IBaseFilter 接口（IbaseFilter 从 IMediaFilter 继承而来）。同时，CBaseFilter 还实现了 Filter 框架（描述了各个 Pin 组件的情况）。

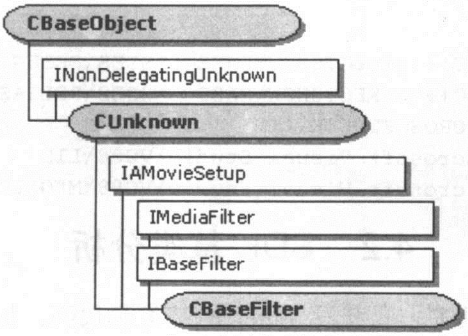

图 4.2　CBaseFilter 类结构

CBaseFilter 类的使用方法如下：

（1）声明一个新类是从 CbaseFilter 中继承而来；
（2）在新类中定义 Filter 上的 Pin 的实例（Pin 从 CBasePin 类继承而来）；
（3）实现纯虚函数 CBaseFilter::GetPin，用于返回 Filter 上各个 Pin 的对象指针；
（4）实现纯虚函数 CBaseFilter::GetPinCount，用于返回 Filter 上 Pin 的数量；
（5）考虑如何处理从输入 Pin 进来的 Sample 数据。

提示

SDK 类中的 CSource、CBaseRenderer、CTransformFilter 都是从 CBaseFilter 继承来的。实际开发 Filter 的时候，使用这些子类作为 Filter 类（或者自定义 Filter 的父类）会更容易。

4.2.4　CBasePin

Filter 上最基本的 Pin 由 CBasePin 类（参见 amfilter.cpp）实现。

如图 4.3 所示，作为 Pin 的基本特征，CBasePin 实现了 IPin 接口。CBasePin 设计了 Pin 的整个连接过程。另外，这个类还实现了 IQualityControl 接口，该接口用于质量控制。

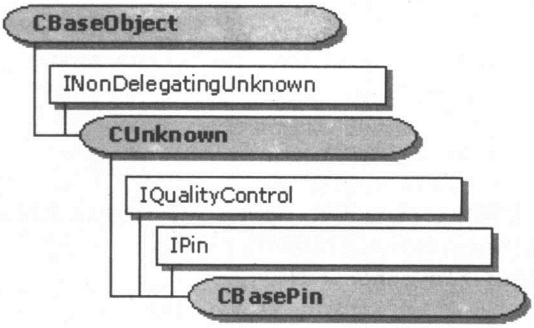

图 4.3　CBasePin 类结构

在 CBasePin 实现的成员函数中，有 3 个与 Filter 的状态转换相对应，见表 4.1。

表 4.1　CBasePin 上调用的函数与 Filter 的状态转换的对应关系

Filter 状态	CBasePin 上调用的函数
停止（Stopped）	CBasePin::Inactive
暂停（Paused）	CBasePin::Active
运行（Running）	CBasePin::Run

如下代码是 CBaseFilter::Stop、CBaseFilter::Pause、CBaseFilter::Run 的实现：

```
// Filter 调用 Stop，实际上就是调用 Filter 的所有 Pin 的 Inactive 函数
STDMETHODIMP CBaseFilter::Stop()
{
    CAutoLock cObjectLock(m_pLock);
    HRESULT hr = NOERROR;
    // notify all pins of the state change
```

```
    if (m_State != State_Stopped) {
        int cPins = GetPinCount();
        for (int c = 0; c < cPins; c++) {
            CBasePin *pPin = GetPin(c);
// 仅在完成连接的 Pin 上调用 Inactive 函数，如果 Inactive 函数返回一个错误值，则暂时
// 忽略，以使所有 Pin 都有机会被调用 Inactive
            if (pPin->IsConnected()) {
                HRESULT hrTmp = pPin->Inactive();
                if (FAILED(hrTmp) && SUCCEEDED(hr)) {
                    hr = hrTmp;
                }
            }
        }
    }
    m_State = State_Stopped;
    return hr;
}
// Filter 调用 Pause，实际上就是调用 Filter 的所有 Pin 的 Active 函数
STDMETHODIMP CBaseFilter::Pause()
{
    CAutoLock cObjectLock(m_pLock);

    // notify all pins of the change to active state
    if (m_State == State_Stopped) {
        int cPins = GetPinCount();
        for (int c = 0; c < cPins; c++) {
            CBasePin *pPin = GetPin(c);
// 仅在完成连接的 Pin 上调用 Active 函数，如果其中有一个 Pin 上的 Active 函数调用
// 失败，则让 CBaseFilter::Pause 直接返回这个错误值
            if (pPin->IsConnected()) {
                HRESULT hr = pPin->Active();
                if (FAILED(hr)) {
                    return hr;
                }
            }
        }
    }
    m_State = State_Paused;
    return S_OK;
}
// Filter 调用 Run，实际上就是调用 Filter 的所有 Pin 的 Run 函数
STDMETHODIMP CBaseFilter::Run(REFERENCE_TIME tStart)
{
    CAutoLock cObjectLock(m_pLock);
    // remember the stream time offset
    m_tStart = tStart;
    // 暂停状态是 Filter 从停止到运行切换的必经状态！
    if (m_State == State_Stopped){
        HRESULT hr = Pause();
        if (FAILED(hr)) {
```

```
                return hr;
            }
    }
    // notify all pins of the change to active state
    if (m_State != State_Running) {
        int cPins = GetPinCount();
        for (int c = 0; c < cPins; c++) {
                CBasePin *pPin = GetPin(c);
                // 仅在完成连接的 Pin 上调用 Run 函数
                if (pPin->IsConnected()) {
                    HRESULT hr = pPin->Run(tStart);
                    if (FAILED(hr)) {
                        return hr;
                    }
                }
            }
        }
        m_State = State_Running;
        return S_OK;
    }
```

在实际开发 Filter 的过程中，很有可能重写 CBasePin::Inactive、CBasePin::Active 和 CBasePin::Run 这 3 个函数，以进行必要的初始化、释放资源等。

CBasePin 类的使用方法如下：

（1）从 CBasePin 派生一个子类；

（2）实现纯虚函数 CBasePin::CheckMediaType，进行 Pin 连接时的媒体类型检查；

（3）实现虚函数 CBasePin::GetMediaType，提供 Pin 上的首选媒体类型；

（4）实现 IPin::BeginFlush 和 IPin::EndFlush 两个函数；

（5）可能需要重写的函数包括 CBasePin::Active（实现资源分配）、CBasePin::Inactive（实现资源释放）、CBasePin::Run（在 Filter 运行前进行一些初始化）、CBasePin::CheckConnect（连接时检查，如查询对方 Pin 上是否支持某个特殊接口）、CBasePin::BreakConnect（断开连接，并进行必要的资源释放）、CBasePin::CompleteConnect（完成连接时被调用，可以在这个函数中获得当前连接用的媒体类型等参数）、CBasePin::EndOfStream（当上流数据全部传送完毕后被调用，如果这是 Transform Filter，则将 EndOfStream 继续往下传送；如果是 Renderer Filter，需要向 Filter Graph Manager 发送一个 EC_COMPLETE 事件）、CBasePin::Notify（直接响应质量控制，或者将质量消息往上一级 Filter 发送）。

4.2.5 CBaseInputPin 和 CBaseOutputPin

从 CBasePin 类派生的，也是很基本的输入或输出 Pin 的类为 CBaseInputPin（参见 amfilter.cpp）和 CBaseOutputPin（参见 amfilter.cpp）。

如图 4.4 所示，CBaseInputPin 实现了 IMemInputPin（用于推模式的数据传送），而 CBaseOutputPin 主要完成了传送数据所使用的 Sample 管理器（Allocator）的协商，并重写了 CBasePin::Active（用于实际的 Sample 内存分配）以及 CBasePin::Inactive（用于

Sample 内存的释放）。

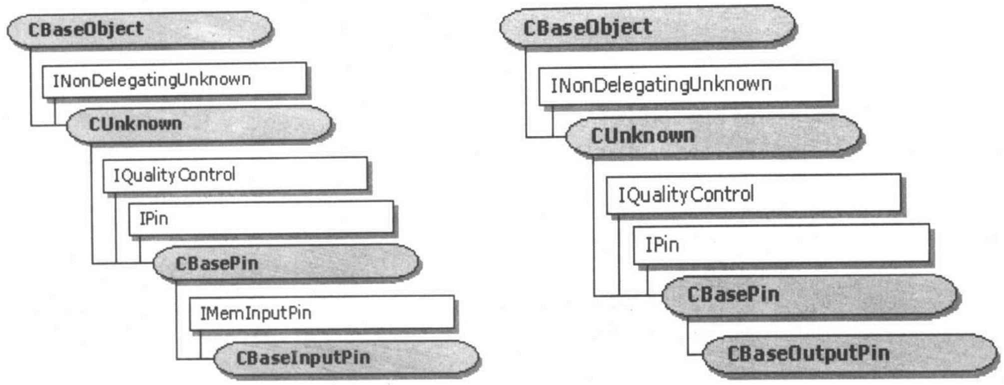

图 4.4 CBaseInputPin 和 CBaseOutputPin 类结构

CBaseInputPin 类的使用方法（派生一个子类，并且最少需要重写以下函数）如下：

（1）CBaseInputPin::BeginFlush

（2）CBaseInputPin::EndFlush

（3）CBaseInputPin::Receive

（4）CBasePin::CheckMediaType

（5）CBasePin::GetMediaType

CBaseOutputPin 类的使用方法（派生一个子类，并且最少需要重写以下函数）如下：

（1）重写 CBasePin::CheckMediaType 进行连接时媒体类型的检查；

（2）实现纯虚函数 CBaseOutputPin::DecideBufferSize，决定 Sample 内存的大小；

（3）重写 CBasePin::GetMediaType，提供 Pin 上的首选媒体类型。

提示 可以看到，输入 Pin 和输出 Pin 都可以通过重写 CBasePin::GetMediaType 来提供 Pin 上的首选媒体类型，但在 Filter 的实际开发中，一般只在输出 Pin 上这么做。

Q 如何直接使用 CBaseFilter、CBaseInputPin、CBaseOutputPin 来开发自己的 Filter?

A 如果开发的是一个 Transform Filter，Filter 的父类很多时候都是选择 CTransform-Filter 或 CTransInPlaceFilter，这种 Filter 的开发相对来说也比较简单。但有些时候，Filter 框架不得不选择 CBaseFilter、CBaseInputPin、CBaseOutputPin 等类来实现，这就有点麻烦了。这时候，可以参考 CTransformFilter、CTransformInputPin、CTransformOutputPin 对上述 3 个基类的使用方法，以此来指导 Filter 的开发。

4.2.6 CSource

DirectShow SDK 还提供了其他的更加实用的 Filter 类，包括 CSource、CTransform-

Filter、CTransInPlaceFilter、CVideoTransformFilter、CBaseRenderer、CBase Video Renderer 等。它们的类继承关系如图 4.5 所示。

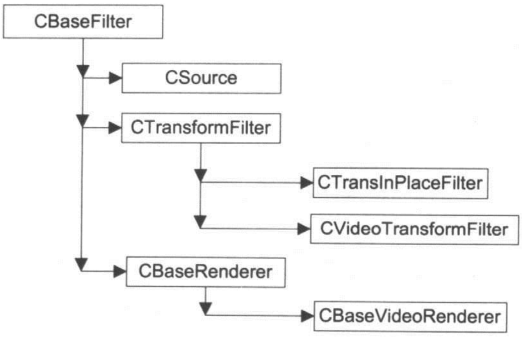

图 4.5　SDK Filter 类继承关系

如图 4.6 所示，CSource 类（参见 source.cpp 的实现）直接从 CBaseFilter 中继承而来，一般作为推模式 Source Filter 的父类。

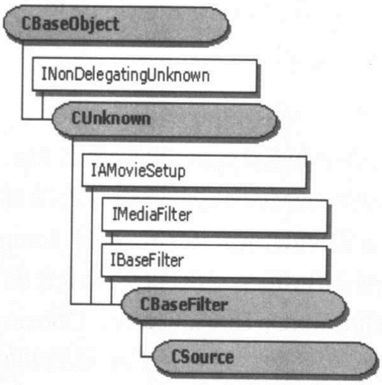

图 4.6　CSource 类结构

CSource 类的使用方法如下：

（1）从 CSource 类中派生一个新的 Filter 类；

（2）在新的 Filter 类的构造函数中创建各个 CSourceStream 类实例（CSourceStream 类的构造函数会自动将该 Pin 加入 Filter 中，并在析构函数中自动删除）；

（3）使用 CSource::pStateLock 函数返回的同步对象进行 Filter 对象上的多线程同步。

提示　使用 CSource 作为 Filter 父类的 Filter 未必都是 Source Filter。在有些开发 Transform Filter 的应用中，输出 Pin 需要使用独立的线程（即与输入 Pin 上传送数据不同的线程）传送数据时，也可以考虑使用 CSource。

4.2.7　CSourceStream

CSource 实际上继承自 CBaseFilter，提供了一个“推”数据的能力。这种 Filter 至少

有一个输出 Pin 采用了 CSourceStream 类（或 CSourceStream 的子类）。如图 4.7 所示，CSourceStream 上实现了一个线程（CSourceStream 从 CAMThread 类继承而来），Sample 数据就是靠这个线程向下一级 Filter 发送的。

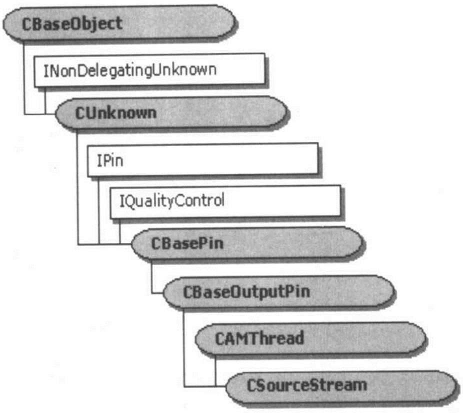

图 4.7　CSourceStream 类结构

CSourceStream 类的使用方法如下：

（1）从 CSourceStream 派生一个输出 Pin 类；

（2）重写 CSourceStream::GetMediaType，提供输出 Pin 上的首选媒体类型；

（3）重写 CSourceStream::CheckMediaType，进行连接时媒体类型的检查；

（4）实现 CBaseOutputPin::DecideBufferSize，决定 Sample 内存的大小；

（5）实现 CSourceStream::FillBuffer，为即将传送出去的 Sample 填充数据；

（6）可选地实现 CSourceStream::OnThreadCreate、CSourceStream::OnThreadDestroy、CSourceStream::OnThreadStartPlay 等函数，进行适当时机的初始化、资源管理等操作。

4.2.8　CTransformFilter

CTransformFilter 类（参见 transfrm.cpp 的实现）是开发 Transform Filter 最基本的类，也是最常用到的类。其结构如图 4.8 所示。它有一个输入 Pin 和一个输出 Pin，分别使用 CTransformInputPin 类和 CTransformOutputPin 类，其结构如图 4.9 所示。

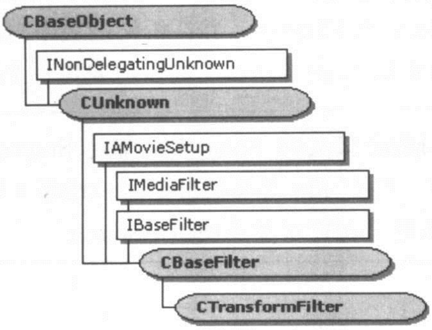

图 4.8　CTransformFilter 类结构

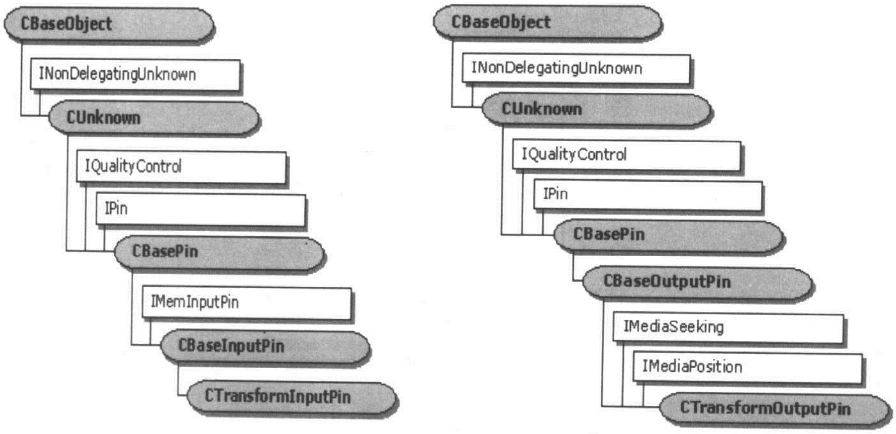

图 4.9　CTransformInputPin 和 CTransformOutputPin 类结构

从图 4.8 和图 4.9 可以看出，CTransformFilter 从 CBaseFilter 继承而来，CTransformInputPin 从 CBaseInputPin 继承而来，CTransformOutputPin 从 CBaseOutputPin 继承而来。另外，在 CTransformOutputPin 上还实现了 IMediaSeeking 和 IMediaPosition 接口（其实，CTransformOutputPin 并没有真正实现各个 Seek 操作。在实际的 Seek 操作请求发生时，CTransformOutputPin 会将这些操作请求转发给上一级 Filter 的输出 Pin）。

CTransformFilter 实现的最大特征是，它将 Pin 上必须实现的函数都“委托”到了 Filter 上（Pin 上必须实现的函数在 Filter 上有类似的函数定义）。一般无需重写输入和输出 Pin 类，而只须在 Filter 上实现相应的函数就行了。

提示　CTransformFilter 默认在 GetPin 函数实现中创建输入和输出 Pin。因此，如果重写了自己的输入或输出 Pin 类，需要同时重写 GetPin 函数。

CTransformFilter 类的使用方法（派生一个 Filter 子类，并且最少需要重写以下函数）如下：

（1）CTransformFilter::CheckInputType
（2）CTransformFilter::CheckTransform
（3）CTransformFilter::DecideBufferSize
（4）CTransformFilter::GetMediaType
（5）CTransformFilter::Transform

4.2.9　CTransInPlaceFilter

CTransInPlaceFilter（参见 transip.cpp 的实现）是一个“就地”处理的 Transform Filter 类。其结构如图 4.10 所示。与 CTransformFilter 相似，CTransInPlaceFilter 也有一个输入 Pin 和一个输出 Pin，但使用 CTransInPlaceInputPin 类和 CTransInPlaceOutputPin 类。其结构如图 4.11 所示。

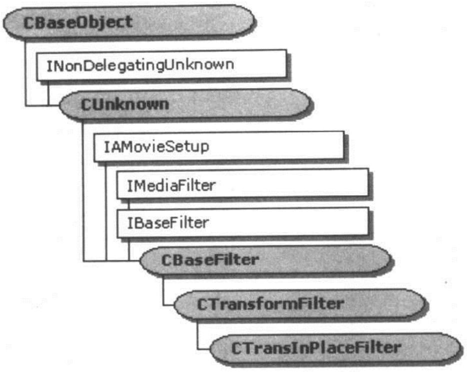

图 4.10　CTransInPlaceFilter 类结构

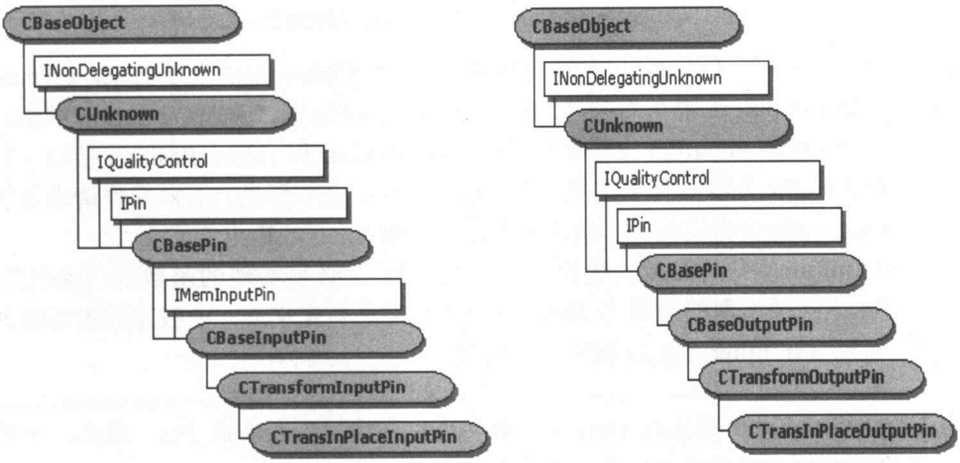

图 4.11　CTransInPlaceInputPin 和 CTransInPlaceOutputPin 类结构

CTransInPlaceFilter 的输入和输出 Pin 上一般使用相同的媒体类型进行连接，并且尽量使用同一个 Sample 管理器（如果 Filter 实现时要修改 Sample 数据，而协商达成一致的 Sample 管理器是只读的，那么 CTransInPlaceFilter 的输入和输出 Pin 将不得不使用各自的 Sample 管理器）。这样可以避免 Sample 数据从输入 Pin 到输出 Pin 的一次内存复制，体现"就地"处理的高效性。

CTransInPlaceFilter 类要实现上述的目标，主要依赖于 CTransInPlaceFilter::CompleteConnect、CTransInPlaceInputPin::GetAllocator 和 CTransInPlaceInputPin::NotifyAllocator 的函数实现，代码如下：

```
// 当输入 Pin 或输出 Pin 完成连接时被调用
// 经过一个反复重连的过程，来达到输入和输出 Pin 使用相同的媒体类型的目的
HRESULT CTransInPlaceFilter::CompleteConnect(PIN_DIRECTION dir,IPin
*pReceivePin)
{
    UNREFERENCED_PARAMETER(pReceivePin);
    ASSERT(m_pInput);
    ASSERT(m_pOutput);
```

```
    // if we are not part of a graph, then don't indirect the pointer
    // this probably prevents use of the filter without a filtergraph
    if (!m_pGraph) {
        return VFW_E_NOT_IN_GRAPH;
    }
    // 当输出 Pin 调用该函数（并且此时输入 Pin 已连上）时，使用输出 Pin 上的媒体类型对
    // 输入 Pin 进行重连接
    if (dir == PINDIR_OUTPUT) {
        if( m_pInput->IsConnected() ) {
            return ReconnectPin( m_pInput, &m_pOutput
            ->CurrentMediaType() );
        }
        return NOERROR;
    }
    ASSERT(dir == PINDIR_INPUT);
    // 当输入 Pin 调用该函数（并且此时输出 Pin 已连上）时，如果输入和输出 Pin 上使用的
    // 媒体类型不一致，则使用输入 Pin 上的媒体类型对输出 Pin 进行重连接
    if( m_pOutput->IsConnected() ) {
        if (  m_pInput->CurrentMediaType()
            != m_pOutput->CurrentMediaType()
            ) {
            return ReconnectPin( m_pOutput, &m_pInput
            ->CurrentMediaType() );
        }
    }
    return NOERROR;
}

// 当上一级 Filter 的输出 Pin 要求我们的输入 Pin 提供 Sample 管理器时，
// 如果我们的输出 Pin 已经连上，则可以取出输出 Pin 上的 Sample 管理器提供给上一级
// Filter，以此来达到我们的输入和输出 Pin 使用同一个 Sample 管理器的目的
STDMETHODIMP CTransInPlaceInputPin::GetAllocator(IMemAllocator **
ppAllocator)
{
    CheckPointer(ppAllocator,E_POINTER);
    ValidateReadWritePtr(ppAllocator,sizeof(IMemAllocator *));
    CAutoLock cObjectLock(m_pLock);
    HRESULT hr;
    if ( m_pTIPFilter->m_pOutput->IsConnected() ) {
        //  Store the allocator we got
        hr = m_pTIPFilter->OutputPin()->ConnectedIMemInputPin()
                                        ->GetAllocator( ppAllocator );
        if (SUCCEEDED(hr)) {
            m_pTIPFilter->OutputPin()->SetAllocator( *ppAllocator );
        }
    }
    else {
        //  Help upstream filter (eg TIP filter which is having to do a
        //  copy)by providing a temp allocator here - we'll never use
        //  this allocator because when our output is connected we'll
```

```
        // reconnect this pin
        hr = CTransformInputPin::GetAllocator( ppAllocator );
    }
    return hr;
}
// 上一级 Filter 调用该函数，告知输入 Pin 上到底使用哪一个 Sample 管理器
// 如果设置进来的 Sample 管理器是只读的，而我们在 Filter 中又要修改数据，
// 则我们的 Filter 不得不最终使用不同的 Sample 管理器
STDMETHODIMP CTransInPlaceInputPin::NotifyAllocator(IMemAllocator *
pAllocator,
    BOOL bReadOnly)
{
    HRESULT hr = S_OK;
    CheckPointer(pAllocator,E_POINTER);
    ValidateReadPtr(pAllocator,sizeof(IMemAllocator));
    CAutoLock cObjectLock(m_pLock);
    m_bReadOnly = bReadOnly;
    // If we modify data then don't accept the allocator if it's
    // the same as the output pin's allocator
    // If our output is not connected just accept the allocator
    // We're never going to use this allocator because when our
    // output pin is connected we'll reconnect this pin
    if (!m_pTIPFilter->OutputPin()->IsConnected()) {
        return CTransformInputPin::NotifyAllocator(pAllocator,
bReadOnly);
    }
    // If the allocator is read-only and we're modifying data
    // and the allocator is the same as the output pin's
    // then reject
    if (bReadOnly && m_pTIPFilter->m_bModifiesData) {
        IMemAllocator *pOutputAllocator =
            m_pTIPFilter->OutputPin()->PeekAllocator();
        // Make sure we have an output allocator
        if (pOutputAllocator == NULL) {
            hr = m_pTIPFilter->OutputPin()->ConnectedIMemInputPin()->
                                    GetAllocator(&pOutputAllocator);
            if(FAILED(hr)) {
                hr = CreateMemoryAllocator(&pOutputAllocator);
            }
            if (SUCCEEDED(hr)) {
                                    m_pTIPFilter->OutputPin()-
              SetAllocator(pOutputAllocator);
                pOutputAllocator->Release();
            }
        }
        if (pAllocator == pOutputAllocator) {
            hr = E_FAIL;
        } else if(SUCCEEDED(hr)) {
            // Must copy so set the allocator properties on the output
            ALLOCATOR_PROPERTIES Props, Actual;
```

```
            hr = pAllocator->GetProperties(&Props);
            if (SUCCEEDED(hr)) {
                hr = pOutputAllocator->SetProperties(&Props, &Actual);
            }
            if (SUCCEEDED(hr)) {
                if (  (Props.cBuffers > Actual.cBuffers)
                   || (Props.cbBuffer > Actual.cbBuffer)
                   || (Props.cbAlign  > Actual.cbAlign)
                   ) {
                    hr =  E_FAIL;
                }
            }
            //  Set the allocator on the output pin
            if (SUCCEEDED(hr)) {
                hr = m_pTIPFilter->OutputPin()->ConnectedIMemInputPin()
                ->NotifyAllocator( pOutputAllocator, FALSE );
            }
        }
    } else {
        hr = m_pTIPFilter->OutputPin()->ConnectedIMemInputPin()
             ->NotifyAllocator( pAllocator,bReadOnly );
        if (SUCCEEDED(hr)) {
            m_pTIPFilter->OutputPin()->SetAllocator( pAllocator );
        }
    }
    if (SUCCEEDED(hr)) {
        // It's possible that the old and the new are the same thing
        // AddRef before release ensures that we don't unload it
        pAllocator->AddRef();
        if( m_pAllocator != NULL )
            m_pAllocator->Release();
        m_pAllocator = pAllocator;    // We have an allocator for the
                                      // input pin
    }

    return hr;
}
```

如何确保 CTransInPlaceFilter 的输入和输出 Pin 使用同一个 Sample 管理器？

CTransInPlaceFilter 类定义了一个成员变量 m_bModifiesData，用于指示我们在 Filter 中是否会修改 Sample 数据。这个变量在 CTransInPlaceFilter 构造函数调用时被默认初始化为 true。如果我们确定不会在 Filter 中修改 Sample 数据，那么，将 m_bModifiesData 设置为 false，可以保证输入和输出 Pin 连接完成后使用同一个 Sample 管理器。

提示

与 CTransformFilter 相似，CTransInPlaceFilter 默认在 GetPin 函数实现中创建输入和输出 Pin。因此，如果我们重写了自己的输入或输出 Pin 类，需要同时重写 GetPin 函数。

CTransInPlaceFilter 类的使用方法（派生一个 Filter 子类，并且最少需要重写以下函数）如下：

（1）CTransformFilter::CheckInputType

（2）CTransInPlaceFilter::Transform

4.2.10 CVideoTransformFilter

CVideoTransformFilter（参见 vtrans.cpp 的实现）是一个实现了视频的质量控制的 Transform Filter 类。其结构如图 4.12 所示。

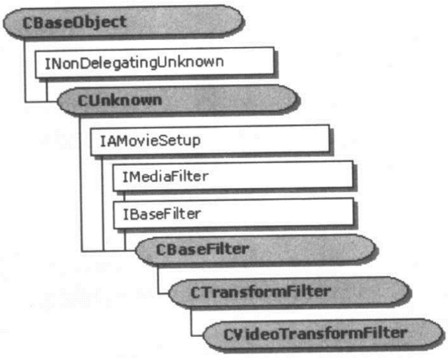

图 4.12 CVideoTransformFilter 类结构

CVideoTransformFilter 通过输入 Pin 上的 Receive 函数接收 Sample 时，能够根据质量消息决定是否丢帧。这个类主要是为开发 AVI 解码 Filter 而设计的。CVideoTransformFilter 类的使用方法基本上与 CTransformFilter 相同，这里不再赘述。

4.2.11 CBaseRenderer

CBaseRenderer（参见 renbase.cpp 的实现）是最基本的实现 Renderer Filter 的类。它默认实现了一个使用 CRendererInputPin 类的输入 Pin。这两个类的结构如图 4.13 所示。

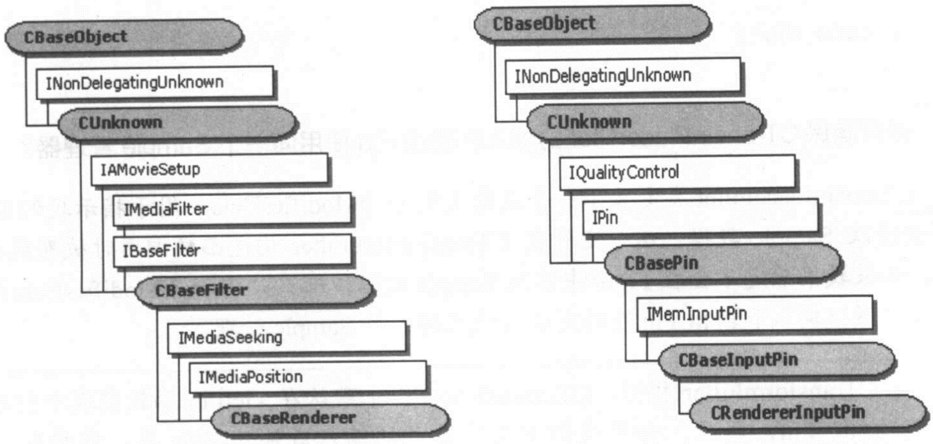

图 4.13 CBaseRenderer 和 CRendererInputPin 类结构

从图中可以看出，CBaseRenderer 从 CBaseFilter 继承而来。另外，CBaseRenderer 上还实现了 IMediaSeeking 和 IMediaPosition 接口。

> **提示**
> 我们曾经在 CTransformOutputPin 上看到过 IMediaSeeking 和 IMediaPosition 这两个接口的实现。比较后不难发现，Transform Filter 在输出 Pin 上实现这两个接口，而 Renderer Filter 在 Filter 上实现这两个接口（因为 Renderer Filter 没有输出 Pin）。

另外，CRendererInputPin 从 CBaseInputPin 继承而来，它把各个主要函数调用都“委托”到 Filter 上。值得注意的是，当输入 Pin 接收到 EndOfStream 调用时，Renderer Filter 有责任向 Filter Graph Manager 发送一个 EC_COMPLETE 事件。

CBaseRenderer 类的使用方法（派生一个子类，并至少实现如下函数）如下：

（1）CBaseRenderer::CheckMediaType，用于检查输入 Pin 连接用的媒体类型；

（2）CBaseRenderer::DoRenderSample，处理当前的 Sample。

> **提示**
> CBaseRenderer 实际上是为用于播放的 Renderer Filter 设计的，对 Sample 的安排比较复杂。如果我们要开发的 Renderer Filter 不播放 Sample（比如写文件的 Filter、或者负责网络的 Filter），Filter 的基类可以选择 CBaseFilter，而此时输入 Pin 最好选择从 CRenderedInputPin 类派生（参见 amextra.cpp 的实现）。

4.2.12　CBaseVideoRenderer

CBaseVideoRenderer（参见 renbase.cpp 的实现）是一个实现 Video Renderer 的基类。其结构如图 4.14 所示。

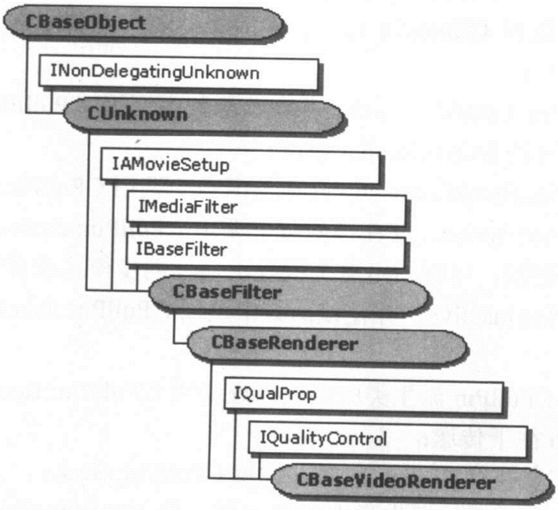

图 4.14　CBaseVideoRenderer 类结构

CBaseVideoRenderer 在 CBaseRenderer 的基础上增加了 IQualityControl 和 IQualProp 接口的实现。其中，IQualityControl 用于视频质量控制，IQualProp 用于在 Filter 属性页显示

一些实时性能参数。CBaseVideoRenderer 类的使用方法基本上与 CBaseRenderer 相同，这里不再赘述。

在 DirectShow SDK 基类库中，除了上述 Filter 和 Pin 类外，还有很多工具类。有了这些类的支持，我们开发 Filter 组件或者 DirectShow 应用程序会更加轻松。这些类主要包括，CPullPin、COutputQueue、CSourceSeeking、CEnumPins、CEnumMediaTypes、CMemAllocator、CMediaSample、CBaseReferenceClock、CMediaType、CBasePropertyPage 等，下面分别介绍。

4.2.13 CPullPin

CPullPin 类（参见 pullpin.cpp 的实现）辅助实现拉模式的输入 Pin。与拉模式的输入 Pin 相连接的上一级的输出 Pin，必然实现了 IAsyncReader 接口。

如图 4.15 所示，CPullPin 并不是从 CBasePin 派生的，所以它不能作为 Pin 直接使用。从 CAMThread 中继承而来，意味着 CPullPin 使用了一个独立的线程去“拉”数据，即调用上一级 Filter 的输出 Pin 上实现的 IAsyncReader 接口方法。

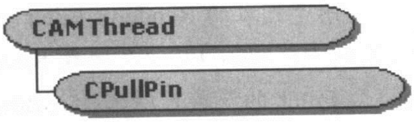

图 4.15 CPullPin 类结构

CPullPin 类的使用方法如下：

（1）从 CBasePin 派生一个类作为输入 Pin 类，从 CPullPin 派生一个新类用于重新实现 CPullPin 的部分函数，然后在 Pin 类中定义一个 CPullPin 类型的成员对象（因为 CPullPin 的部分成员函数和 CBasePin 已经实现的成员函数命名冲突，所以 CPullPin 不能直接作为 Pin 类的父类）；

（2）重写 CBasePin::CheckConnect，并在该函数中调用 CPullPin::Connect，用于 Pin 连接的时候查询对方 Pin 的 IAsyncReader 接口；

（3）重写 CBasePin::BreakConnect，并在该函数中调用 CPullPin::Disconnect；

（4）重写 CBasePin::Active，并在该函数中调用 CPullPin::Active，用于启动“拉”数据线程（当 Pin 连接的时候，可以指定这个线程工作在同步模式或者异步模式）；

（5）重写 CBasePin::Inactive，并在该函数中调用 CPullPin::Inactive，用于停止“拉”数据线程；

（6）在新生成的 CPullPin 派生类中实现纯虚函数 CPullPin::Receive 处理 Sample，并将 Sample 通过输出 Pin 往下传送；

（7）当 Filter 上需要执行 Seek 操作时，调用 CPullPin::Seek；

（8）在新生成的 CPullPin 派生类中实现纯虚函数 CPullPin::EndOfStream、CPullPin::BeginFlush、CPullPin::EndFlush；

（9）在新生成的 CPullPin 派生类中实现纯虚函数 CPullPin::OnError 进行错误处理。

4.2.14 COutputQueue

COutputQueue 类（参见 outputq.cpp 的实现）实现了 Sample 的队列式传送，它可以工作在同步模式，也可以工作在异步模式（使用一个独立的线程）。其结构如图 4.16 所示。

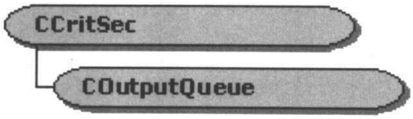

图 4.16 COutputQueue 类结构

COutputQueue 类的使用方法如下：

（1）在输出 Pin 上创建 COutputQueue 类的对象实例，并在 COutputQueue 的构造函数中指定这个输出 Pin 连接着的输入 Pin 指针；

（2）将以往输出 Pin 调用连接着的输入 Pin 上的函数，改成调用 COutputQueue 对象的相应函数，这些函数对应关系见表 4.2。

表 4.2 函数对应关系

本该在输入 Pin 上调用的函数	现在 COutputQueue 对象上调用的相应函数
IPin::BeginFlush	BeginFlush
IPin::EndFlush	EndFlush
IPin::EndOfStream	EOS
IPin::NewSegment	NewSegment
IMemInputPin::Receive	Receive
IMemInputPin::ReceiveMultiple	ReceiveMultiple

（3）更多的使用细节参见 SDK 目录中的 Filter 例子 inftee。

4.2.15 CSourceSeeking

CSourceSeeking 类（参见 ctlutil.cpp 的实现）真正实现了 IMediaSeeking 的各个接口方法，在我们开发推模式的 Source Filter 并支持 Seek 操作时特别有用。其结构如图 4.17 所示。

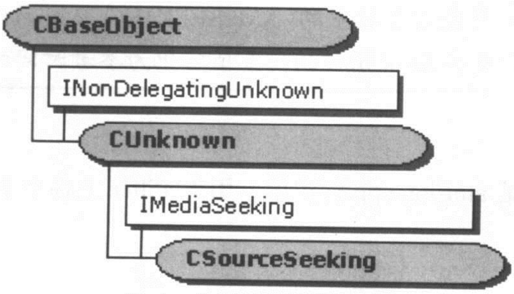

图 4.17 CSourceSeeking 类结构

默认情况下，CSourceSeeking 只支持 TIME_FORMAT_MEDIA_TIME 这一种格式的 Seek 操作。

CSourceSeeking 类的使用方法（假设我们的 Filter 从 CSource 继承而来，输出 Pin——CMyStream 同时从 CSourceStream 和 CSourceSeeking 继承而来）如下：

（1）实现纯虚函数 CSourceSeeking::ChangeStart，该函数在开始位置改变时被调用；

（2）实现纯虚函数 CSourceSeeking::ChangeStop，该函数在结束位置改变时被调用；

（3）实现纯虚函数 CSourceSeeking::ChangeRate，该函数在播放速率改变时被调用；

（4）在 CMyStream 实现如下一个成员函数 UpdateFromSeek:

```
void CMyStream::UpdateFromSeek()
{
    if (ThreadExists())
    {
        DeliverBeginFlush();    // 从输出 Pin 上发送 BeginFlush
        Stop();                 // 停止输出 Pin 上的数据发送线程
        DeliverEndFlush();      // 从输出 Pin 上发送 EndFlush
        Run();                  // 重新启动数据发送线程
    }
}
```

每次 Seek 操作（改变了开始位置、结束位置、播放速率等）后，调用 UpdateFromSeek 函数。

（5）重写 CSourceStream::OnThreadStartPlay，代码如下：

```
HRESULT CMyStream::OnThreadStartPlay()
{
    m_bDiscontinuity = TRUE;
    return DeliverNewSegment(m_rtStart, m_rtStop, m_dRateSeeking);
}
```

（6）如果要支持 TIME_FORMAT_MEDIA_TIME 外的更多格式，需要重新实现 IMediaSeeking::GetTimeFormat、IMediaSeeking::IsFormatSupported、IMediaSeeking::IsUsingTimeFormat、IMediaSeeking::QueryPreferredFormat、IMediaSeeking::SetTimeFormat 等函数。

提示 在实现 TIME_FORMAT_MEDIA_TIME 外的其他格式时，IMediaSeeking::SetPositions 函数的实现需要注意查看参数中的 AM_SEEKING_ReturnTime 标记。如果参数中含有这个标记，则要将 Seek 操作的位置值转化为参考时间的格式返回。

4.2.16 CEnumPins

CEnumPins 类（参见 amfilter.cpp 的实现）用于 Filter 上各个 Pin 的枚举。其结构如图 4.18 所示。

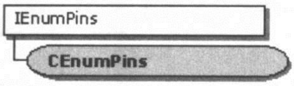

图 4.18 CEnumPins 类结构

CEnumPins 实现了 IEnumPins 接口。在 CEnumPins 类的内部实现中，调用了 Filter 函

数 CBaseFilter::GetPin、CBaseFilter::GetPinCount 和 CBaseFilter::GetPinVersion。如果 Filter 的 Pin 是动态创建或删除的，此时 Filter 一般会调用 IncrementPinVersion 函数改变 Pin 的版本号。枚举器会检测到这个版本变化，返回一个错误值 VFW_E_ENUM_OUT_OF_SYNC 表示“不同步”。此时，调用者应该调用 CEnumPins::Reset 复位枚举器的内部状态。

Q 如何枚举 Filter 上的所有 Pin（返回一个输入或输出 Pin）？

A 实现代码如下:

```
IPin *GetPin(IBaseFilter *pFilter, PIN_DIRECTION PinDir)
{
    BOOL        bFound = FALSE;
    IEnumPins   *pEnum;
    IPin        *pPin;

    HRESULT hr = pFilter->EnumPins(&pEnum);
    if (FAILED(hr))
    {
        return NULL;
    }
    while(pEnum->Next(1, &pPin, 0) == S_OK)
    {
        PIN_DIRECTION PinDirThis;
        pPin->QueryDirection(&PinDirThis);
        if (bFound = (PinDir == PinDirThis))
            break;
        pPin->Release();
    }
    pEnum->Release();
    return (bFound ? pPin : NULL);
}
```

4.2.17 CEnumMediaTypes

CEnumMediaTypes 类（参见 amfilter.cpp）用于 Pin 上的首选媒体类型枚举。其结构如图 4.19 所示。

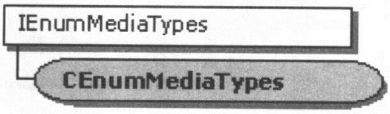

图 4.19　CEnumMediaTypes 类结构

CEnumMediaTypes 实现了 IEnumMediaTypes 接口。在 CEnumMediaTypes 类的内部实现中，调用了 Pin 上的函数 CBasePin::GetMediaType 和 CBasePin::GetMediaTypeVersion。如果 Pin 上提供的首选媒体类型会动态改变，此时该 Pin 一般会调用 IncrementTypeVersion 函数改变首选媒体类型的版本号。枚举器会检测到这个版本变化，返回一个错误值

VFW_E_ENUM_OUT_OF_SYNC 表示“不同步”。此时，调用者应该调用 CEnumMediaTypes::Reset 复位枚举器的内部状态。

Q 如何枚举 Pin 上的所有首选媒体类型？

A 实现代码如下：

```
BOOL GetMediaType(IPin * inPin)
{
	IEnumMediaTypes * pMTEnum = NULL;
	AM_MEDIA_TYPE *   pAMType = NULL;
	if (SUCCEEDED(inPin->EnumMediaTypes(&pMTEnum)))
	{
		ASSERT(pMTEnum);
		while (pMTEnum->Next(1, &pAMType, 0) == S_OK)
		{
			// 对取得的媒体进行处理（保存或者返回）...
			DeleteMediaType(pAMType);
		}
		pMTEnum->Release();
		return TRUE;
	}
	return FALSE;
}
```

4.2.18 CMemAllocator

CMemAllocator 类（参见 amfilter.cpp 的实现）实现了使用本地内存传送数据的 Sample 管理器。其结构如图 4.20 所示。

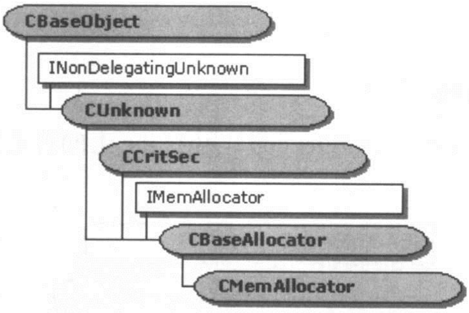

图 4.20 CMemAllocator 类结构

CMemAllocator 的父类为 CBaseAllocator，并且实现了 IMemAllocator 接口。Sample 管理器用于提供 Filter 传送数据所使用的 Sample，包括分配/释放 Sample 的内存、维持 Sample 队列（如果一个管理器申请了多个 Sample 的话）等。

CMemAllocator 的使用方法如下：

（1）调用 CBaseAllocator::SetProperties 设置 Sample 管理器上的属性，包括使用的

Sample 内存大小、Sample 的数量等；

（2）调用 CBaseAllocator::Commit 进行实际 Sample 内存的分配；

（3）调用 CBaseAllocator::GetBuffer 得到一个空 Sample，用于填充数据后传送（一般为阻塞调用，即如果当前没有空 Sample 的话，一直等待）；

（4）Sample 传送出去后，调用 Sample 上的 Release 函数（Sample 也是一种 COM 组件）。值得注意的是，当一个 Sample 的引用计数变成 0 时，Sample 对象没有被析构掉，而是回到了 Sample 管理器上的空闲 Sample 队列，等待下一次被使用；

（5）当不再需要传送数据时，可以调用 CBaseAllocator::Decommit 释放所有 Sample 内存。

4.2.19 CMediaSample

CMediaSample（参见 amfilter.cpp 的实现）是用于数据传送的 Sample 的实现类。其结构如图 4.21 所示。

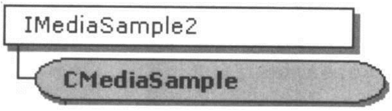

图 4.21 CMediaSample 类结构

CMediaSample 类实现了 IMediaSample2 接口。CMediaSample 封装了一个指向一块数据内存的指针（通过调用 CMediaSample::GetPointer 可以得到），以及其他 Sample 相关的各种属性，包括 Sample 的内存大小、实际数据大小、时间戳、媒体类型等。

提示 IMediaSample2 接口是从 IMediaSample 中继承而来的。IMediaSample2 增加了一次性设置/取出所有属性的接口方法。

Sample 虽然也是一个 COM 组件，但它的生命周期与普通的 COM 组件有些不一样。

（1）在实际应用中，Sample 由 Pin 上的 Sample 管理器创建，并维持在一个空闲 Sample 队列中。当 Filter 要使用一个 Sample 时，会调用 IMemAllocator::GetBuffer 从上述队列中取出一个空 Sample，此时这个 Sample 的引用计数会增加 1；

（2）当 Filter 使用完这个 Sample 后，调用 Sample 对象的 Release 函数。如果这个 Sample 上的引用计数变成 0，它不会像一般的 COM 组件一样析构自己，而是调用 Sample 管理器的 IMemAllocator::ReleaseBuffer 函数，把自己重新加入到空闲 Sample 队列中，代码如下：

```
// Sample 对象上的 Release 函数实现
STDMETHODIMP_(ULONG) CMediaSample::Release()
{
    /* Decrement our own private reference count */
    LONG lRef;
    if (m_cRef == 1) {
        lRef = 0;
```

```
        m_cRef = 0;
    } else {
        lRef = InterlockedDecrement(&m_cRef);
    }
    ASSERT(lRef >= 0);
    DbgLog((LOG_MEMORY,3,TEXT("    Unknown %X ref-- = %d"),
        this, m_cRef));
    /* Did we release our final reference count */
    if (lRef == 0) {
        /* Free all resources */
        if (m_dwFlags & Sample_TypeChanged) {
            SetMediaType(NULL);
        }
        ASSERT(m_pMediaType == NULL);
        m_dwFlags = 0;
        m_dwTypeSpecificFlags = 0;
        m_dwStreamId = AM_STREAM_MEDIA;
        /* This may cause us to be deleted */
        // Our refcount is reliably 0 thus no-one will mess with us
        m_pAllocator->ReleaseBuffer(this);
    }
    return (ULONG)lRef;
}
```

（3）直到 Sample 管理器调用 IMemAllocator::Decommit 时，Sample 对象才真正被析构掉。

4.2.20 CBaseReferenceClock

CBaseReferenceClock（参见 refclock.cpp 的实现）是实现参考时钟的基类。其结构如图 4.22 所示。

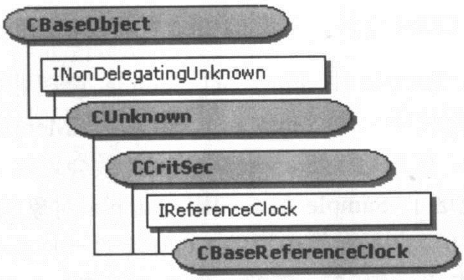

图 4.22　CBaseReferenceClock 类结构

CBaseReferenceClock 实现了 IReferenceClock 接口。一般是 Filter Graph 中的某个 Filter 对象提供整个 Filter Graph 使用的参考时钟，这个 Filter 通常有访问硬件的能力（而且硬件上有定时器模块），那么我们可以使用 CBaseReferenceClock 类来实现自己的参考时钟。

4.2.21　CMediaType

CMediaType（参见 mtype.cpp 的实现）是媒体类型数据结构的封装类。其结构如图 4.23 所示。

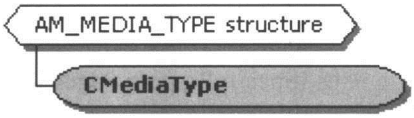

图 4.23　CMediaType 类结构

CMediaType 类从数据结构 AM_MEDIA_TYPE 中继承而来。CMediaType 同时提供了一系列操作媒体类型的非常简便的方法。

4.2.22　CBasePropertyPage

CBasePropertyPage 类（参见 cprop.cpp 的实现）是实现 Filter 的属性页的基类。其结构如图 4.24 所示。

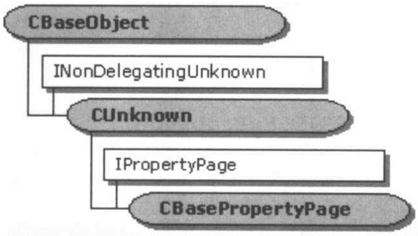

图 4.24　CBasePropertyPage 类结构

CBasePropertyPage 类实现了 IPropertyPage 接口。属性页是一个 COM 组件，因此需要分配一个 CLSID，并且在类工厂中指定一个入口函数。代码如下：

```
CFactoryTemplate g_Templates[] =
{
    {
        L"My Property Page",
        &CLSID_MyPropPage,
        CMyProp::CreateInstance,
        NULL,
        NULL
    },
    /* Also include the template for your filter (not shown). */
};
```

在实现属性页时，我们需要使用 VC 的资源编辑器生成一个对话框（对话框属性中 Style 选 Child，Border 选 None，并且去掉 Title bar 的选中状态），并定义一个字符串 ID 作为对话框窗口的标题；然后从 CBasePropertyPage 类派生一个子类，在构造函数中指定这个对话框的 ID 和字符串 ID。在子类中一般还需实现如下函数：

（1）OnReceiveMessage，处理属性页对话框消息；
（2）OnConnect，一般用于从 Filter 对象上获得某个接口；
（3）OnDisconnect，释放从 Filter 对象上获得的接口；
（4）OnActivate，属性页对话框被激活，需要显示一些属性；
（5）OnApplyChanges，属性页对话框上按下了 Apply 按钮，进行参数的设置。

最后，不要忘记在 Filter 上实现 ISpecifyPropertyPages 接口，步骤如下：

（1）将 Filter 类从 ISpecifyPropertyPages 中继承下来；
（2）在 Filter 上实现 ISpecifyPropertyPages 的接口方法 GetPages，代码如下：

```
STDMETHODIMP CMyFilter::GetPages(CAUUID *pPages)
{
    if (!pPages) return E_POINTER;

    pPages->cElems = 1;
    pPages->pElems = reinterpret_cast<GUID*>(CoTaskMemAlloc
                     (sizeof(GUID)));
    if (pPages->pElems == NULL)
    {
        return E_OUTOFMEMORY;
    }
    *(pPages->pElems) = CLSID_MyPropPage;
    return S_OK;
}
```

（3）在 Filter 的 NonDelegatingQueryInterface 函数实现中“暴露”ISpecifyProperty-Pages 接口，代码如下：

```
STDMETHODIMP CMyFilter::NonDelegatingQueryInterface(REFIID riid, void **
ppv)
{
    CheckPointer(ppv, E_POINTER);

    if (riid == IID_ISpecifyPropertyPages)
    {
            return GetInterface((ISpecifyPropertyPages *) this, ppv);
    }
    else
    {
            return CBaseFilter::NonDelegatingQueryInterface(riid, ppv);
    }
}
```

4.3 Filter 项目的功能分析

原则上，Filter 应该是一个功能单一的模块。如果总的功能需求过于庞大，则需要进行分解。比如项目要求将一路视频、一路音频分别压缩成 MPEG 格式，然后合成 MPEG 流，最终写到一个文件中。最糟糕的设计如图 4.25 所示。

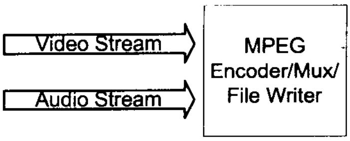

图 4.25　最糟糕的 MPEG 合成 Filter 设计

上述 Filter 要完成 MPEG 音频的编码、视频的编码与音视频的合成，最后再把 MPEG 合成流写入到指定的文件中。Filter 功能如此之复杂，扩展性如此之差，真是有点“辜负”了 DirectShow 架构的设计初衷。

我们推荐的一种设计如图 4.26 所示。

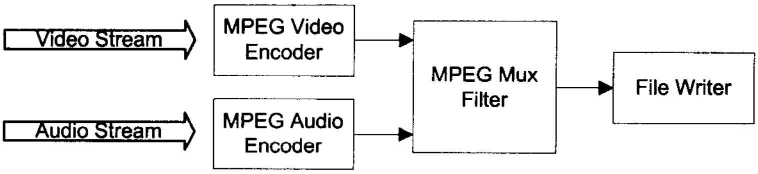

图 4.26　推荐的 MPEG 合成 Filter 设计

每个 Filter 的功能都很单一。在实践中，这样的单个 Filter 更容易调试，总的开发任务也更容易在团队成员中分配。

所以说，Filter 项目的功能分析很重要。如果因为功能分析不当，造成结构设计不理想，轻则影响到 Filter 的扩展性，重则影响到整个项目的进度。下面，我们将结合实践，给出这个功能分析的一般过程。

4.3.1　功能分析的一般过程

分析一个 Filter 的功能，我们主要从以下几个方面考虑：

功能单一化

单个 Filter 的功能不宜太复杂。如果需求过多，可以的话，一般应该分解成几个 Filter 来实现。当然，像上述 MPEG 编码合成项目的两种设计在实际中都是可行的，但两者之间存在一个优劣的问题。

总的来说，Filter 的功能是分离还是集成，应该根据实际情况，坚持“适度”的原则来决定。也不能为了追求功能单一，将功能划分得过细。比如有些 Filter 需要在输入 Pin 上对数据进行一定量的缓存，一般情况下，我们就没有必要将这个缓存功能独立出去而专门写一个 Filter。

选择一种 Filter 模型

我们已经知道，Filter 可以有两种工作模式：推模式和拉模式。但大多数情况下，一个 Filter 只支持一种工作模式。（DirectShow 提供的 MPEG-2 Demultiplexer 是个特例，一般它工作在推模式下，完成 MPEG-2 传输流或程序流的分离；而在 Windows XP 下，这个 Filter 也可以工作在拉模式下，用于 MPEG-2 文件的回放。）

决定使用哪一种工作模式，主要看它参与在 Filter Graph 中工作的位置。大多数情况

下，采用推模式。但当我们的 Filter 必须从前面的“被动源”（比如 File Source）中“拉”数据时，我们就要考虑在 Filter 的输入 Pin 上实现“拉”的功能了。

提示 *即使我们的 Filter 实现了两种工作模式，但在实际应用中，一个 Filter 只能工作在一种模式下。当 Filter 的输入 Pin 连接完成后，Filter 的工作模式一般也就确定了。*

除了工作模式外，我们还可以根据 Filter 要完成的功能，大致决定 Filter 的类型。如果我们的 Filter 不需要数据输入，并且需要将数据主动“推”出去，那么这是一个推模式的 Source Filter；如果我们的 Filter 接收输入数据，然后经过一定的处理后要再输出数据（给别的 Filter 继续处理），那么这是一个 Transform Filter（我们还要考虑是否可以采用“就地”处理方式）；如果我们的 Filter 接收输入数据后不再需要输出，那么这是一个 Renderer Filter。

定义输入和输出

Filter 总是有数据输入或者数据输出，或者输入以后同时也要求有输出。我们必须定义这些输入和输出，包括输入 Pin 的数量、输出 Pin 的数量、各个 Pin 上支持的媒体类型等。当然，根据实际的需要，这些 Pin 和媒体类型都有可能是动态的。

接口定义

接口是 Filter 与外部调用者之间的“桥梁”。外部调用者通过接口可以得到 Filter 的一些属性、状态等参数，也可以控制 Filter 的行为。根据实际需要，我们要考虑定义我们的接口方法，即 Filter 中哪些属性是需要被外部调用者知道的，哪些是可以定制的等。

其他一些特殊需求

Filter 的具体应用是多样化的，所以我们可能还要考虑其他特殊需求。比如在有些 Filter 应用中，输入和输出的 Sample 不是一一对应的，这时就要考虑在输入 Pin 上对数据进行缓存，并且要考虑如何给输出的 Sample 打时间戳等；再比如有些 Filter 内部使用了另外的线程处理数据，我们就要考虑如何让这个线程与 Filter 状态转换同步等问题。

4.3.2 字符叠加 Filter 之功能分析

以开发一个“字符叠加 Filter”为例，我们来具体化上述功能分析的过程。

这个 Filter 的总体需求是，在视频流指定的一个时间段内叠加一段字符，要求能够做到实时叠加。字符的内容、字体类型、大小、颜色、叠加的位置、存在的时间都能够定制，普遍支持 RGB 各种格式（包括 RGB32、RGB24、RGB565、RGB555 等）输入。另外，除了支持静态字符叠加外，还要求支持其他效果的字符（包括滚动字幕、系统时间等）叠加，并且系统设计的扩展性良好。

功能单一化

我们看到，上述的功能总的来说就是一个字符叠加的实现。所有的工作完全可以在一个 Filter 中完成。

选择一种 Filter 模型

这个 Filter 同时有输入和输出（即在输入的视频帧上叠加完字符后再输出到下一级 Filter），因此我们的 Filter 是一个 Transform Filter。由于输入和输出的媒体类型一致，而且输入和输出 Sample 的数据大小不会变化，所以我们可以采用“就地”处理的 Transform Filter。

定义输入输出

我们的 Filter 需要一个输入 Pin 和一个输出 Pin。输入 Pin 支持的媒体类型有 RGB32、RGB24、RGB565、RGB555 等，并且输出 Pin 上的媒体类型与输入 Pin 保持一致。

接口定义

接口方法应该包括设置字符的内容、设置字体的各种属性、选择字符的叠加效果、设置叠加位置、设置字符存在的时间段等。

其他一些特殊需求

我们的 Filter 处理的 Sample 是一一对应的，而且内部不需要另外使用线程。这是一个标准的 Transform Filter。

4.4　Filter 的设计

任何软件开发都应该有一个设计过程，我们的 Filter 开发也不例外。对于同样的功能，好的设计和差的设计相差很大。我们极力反对一拿到需求，不做任何分析（或只做很简单的分析）就开始写代码的做法。

Filter 的设计主要包括以下两方面：选择一个合适的父类和应用结构设计。

4.4.1　选择一个合适的父类

开发一个 Filter，选择一个合适的父类是至关重要的。在 4.2 节，我们已经对 DirectShow SDK 提供的基类结构进行了比较详细的分析。这将有助于我们为自己的 Filter 选择一个合适的父类。

有人说，CBaseFilter 是 Filter 的最上层的基类，选择这个类作为我们 Filter 的父类总是不会错的。这种说法不错，但也不尽然。其实，在实际开发中，我们很少直接使用 CBaseFilter 类。因为 DirectShow SDK 根据各种不同的应用，已经为我们从 CBaseFilter 派生了很多子类，包括 CSource、CTransformFilter、CTransInPlaceFilter、CVideoTransformFilter、CBaseRenderer、CBaseVideoRenderer 等。这些子类已经足够用了。通常，我们只需根据实际的需要，选择其中的一个作为我们 Filter 的父类就行了。

- 当我们要写一个 Source Filter，并且要将产生的数据“推”出去时，我们一般选择 CSource（DirectShow 提供了一个标准的拉模式 Source Filter，一般不用再开发）；
- 我们的 Filter 接收输入数据，经过一定的处理后要再输出数据，我们一般选择 CTransformFilter（如果可以“就地”处理就选择 CTransInPlaceFilter）；

- 如果我们的 Filter 同时有输入和输出，而且它的输入 Pin 上需要自己“拉”数据，则我们选择 CBaseFilter，而在输入 Pin 上使用工具类 CPullPin；
- 当我们的 Filter 接收输入数据后不再需要输出，我们一般选择 CBaseFilter（CBaseRenderer 主要用于 Sample 播放的 Renderer Filter 类，DirectShow 提供了标准的 Video Renderer 或 Audio Renderer。在实际应用中，我们一般不用再开发这种 Filter）。

Filter 的父类确定了，一般 Pin 的父类也就相应地确定了。DirectShow SDK 提供的这些具体应用的 Filter 子类都提供了相应的 Pin 类。我们可以从这些特定的 Pin 类中派生出我们自己的 Pin 类（有些情况，如 Filter 是从 CTransformFilter、CTransInPlaceFilter 等继承而来的，根据实际应用，Pin 类不一定要重新实现）。

以“字符叠加 Filter”为例，我们选择 CTransInPlaceFilter 作为我们 Filter 的父类，而且我们不用重新实现自己的 Pin 类。

4.4.2 应用结构设计

说白了，Filter 只是一个应用框架，一个能够支持“输入→内部处理→输出”的典型过程的应用框架。在这个框架之下，更重要的是应用逻辑（或者说商业逻辑或控制逻辑等），即“内部处理”部分。我们建议将应用框架与应用逻辑分离，以设计一个结构清晰、扩展性良好的 Filter 项目。如图 4.27 所示。

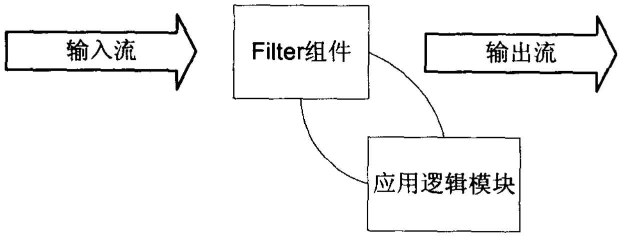

图 4.27 Filter 框架与应用逻辑分离结构

Filter 框架是指完成 Filter 功能的一个必要的操作函数集。在 4.2 节已经介绍了各个 Filter 类的使用方法，包括实现一些必须实现的纯虚函数、可选实现的虚函数等，以完成输入和输出 Pin 上的媒体类型检查，决定 Sample 内存大小等。另外，我们在 Filter 上可能还要将一些状态信息设置给应用逻辑控制对象，在 Filter 的状态转换的同时调用控制对象相应的函数（用于初始化或反初始化），以及在数据处理时调用控制对象的数据处理函数等。

应用逻辑是指对输入数据进行处理的一套控制策略。至于数据是如何输入的，处理完后又是如何输出的，这些问题都由 Filter 框架负责。应用逻辑需要关心的，更多的是数据处理的算法实现、效率优化等问题。应用逻辑控制对象一般以 Filter 的一个复合成员对象的形式出现。

以“字符叠加 Filter”为例，我们定义一个 Filter 类为 CFilterTitleOverlay，它从 CTransInPlaceFilter 类派生而来，再定义一个应用逻辑控制类为 COverlayController，作为 Filter 的一个成员对象（指针），代码如下：

```
class CFilterTitleOverlay : public CTransInPlaceFilter
{
private:
    OVERLAY_TYPE             mOverlayType;
    COverlayController *     mOverlayController;
    // ...
}
```

COverlayController 是最基本的控制类，为了支持多种效果的字符叠加，我们可以从这个类派生更多的子类，如图 4.28 所示。

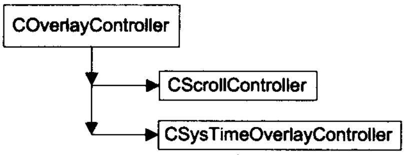

图 4.28　字符叠加应用逻辑控制类结构

其中，CScrollController 用于滚动字幕叠加控制；CSysTimeOverlayController 用于系统时间的叠加控制。

另外，我们还看到 Filter 类中定义了一个枚举类型 OVERLAY_TYPE 的成员变量 mOverlayType，它指示了当前 Filter 使用的叠加方式。OVERLAY_TYPE 的定义如下：

```
typedef enum
{
    OT_NONE,
    OT_STATIC,          // 静态字符叠加
    OT_SYSTIME,         // 系统时间叠加
    OT_SCROLL_TOP,      // 滚动字幕叠加（在视频帧的顶上）
    OT_SCROLL_BOTTOM    // 滚动字幕叠加（在视频帧的底边）
} OVERLAY_TYPE;
```

根据 mOverlayType 的不同取值，我们可以创建不同的控制类对象实例。

4.5　编码实现

Filter 的编码实现包括 Filter 的注册信息、Filter 上的框架函数实现、逻辑控制类实现、自定义接口实现、属性页实现、产权保护等。

> **提示**　下载地址（见前言）中的 FilterTitleOverlay 项目，提供了“字符叠加 Filter”的完整实现。读者在学习下面内容的时候，可以参考这个项目的源代码实现。

4.5.1　Filter 注册信息

Filter 使用前必须先注册。注册信息的填写对于大部分 Filter 开发人员来说都是大同小异的。下面结合“字符叠加 Filter”实例，讲解这个注册信息填写的一般过程。

首先，我们需要为 Filter 生成一个新的 CLSID。如果 Filter 上还实现了属性页、自定

义接口，则还需要分别为它们生成一个 GUID。方法为运行系统中的 GuidGen.exe 程序，出现的界面如图 4.29 所示。

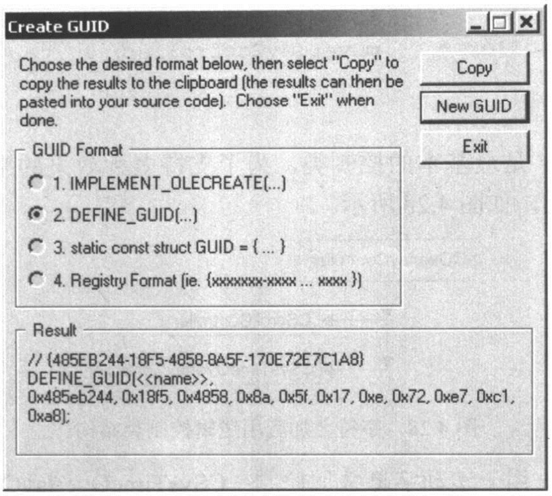

图 4.29　GuidGen.exe 程序界面

单击 New GUID 按钮，就生成一个新的 GUID；单击 Copy 按钮即可以把新生成的 GUID 复制到剪贴板。然后，我们在 VC 源文件的空白地方单击鼠标右键，选择菜单项 Paste，即可以得到 GUID 字符串，形式如下：

```
// {E3FB4BFE-8E5C-4aec-8162-7DA55BE486A1}
DEFINE_GUID(<<name>>,
0xe3fb4bfe, 0x8e5c, 0x4aec, 0x81, 0x62, 0x7d, 0xa5, 0x5b, 0xe4, 0x86,
0xa1);
```

我们再把其中的<<name>>替换成自己设置的名字，代码如下：

```
// {E3FB4BFE-8E5C-4aec-8162-7DA55BE486A1}
DEFINE_GUID(CLSID_HQTitleOverlay,
0xe3fb4bfe, 0x8e5c, 0x4aec, 0x81, 0x62, 0x7d, 0xa5, 0x5b, 0xe4, 0x86,
0xa1);
```

提示　在上述 GUID 字符串之前，需要加入#include <initguid.h>，将其转化为 GUID 常量。

然后，我们就要填写 Filter 的注册信息，代码如下：

```
// setup data
const AMOVIESETUP_MEDIATYPE sudPinTypes =
{
    &MEDIATYPE_NULL,            // Major type
    &MEDIASUBTYPE_NULL          // Minor type
};
const AMOVIESETUP_PIN psudPins[] =
{
    {
```

```
        L"Input",           // String pin name
        FALSE,              // Is it rendered
        FALSE,              // Is it an output
        FALSE,              // Allowed none
        FALSE,              // Allowed many
        &CLSID_NULL,        // Connects to filter
        L"Output",          // Connects to pin
        1,                  // Number of types
        &sudPinTypes },     // The pin details
      { L"Output",          // String pin name
        FALSE,              // Is it rendered
        TRUE,               // Is it an output
        FALSE,              // Allowed none
        FALSE,              // Allowed many
        &CLSID_NULL,        // Connects to filter
        L"Input",           // Connects to pin
        1,                  // Number of types
        &sudPinTypes        // The pin details
    }
};

const AMOVIESETUP_FILTER sudFilter =
{
    &CLSID_HQTitleOverlay,         // Filter CLSID
    L"HQ Title Overlay Std.",      // Filter name
    MERIT_DO_NOT_USE,              // Its merit
    2,                             // Number of pins
    psudPins                       // Pin details
};
// List of class IDs and creator functions for the class factory. This
// provides the link between the OLE entry point in the DLL and an
// object being created. The class factory will call the static
// CreateInstance
CFactoryTemplate g_Templates[] =
{
    {
        L"HQ Title Overlay Std.",
        &CLSID_HQTitleOverlay,
        CFilterTitleOverlay::CreateInstance,
        NULL,
        &sudFilter
    },
    {
        L"HQ Title Overlay Property Page",
        &CLSID_HQTitleOverlayProp,
        CTitleOverlayProp::CreateInstance
    }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

我们看到，我们的 Filter 注册名字为 HQ Title Overlay Std.，有一个输入 Pin 和一个输出 Pin，Filter 的 Merit 值为 MERIT_DO_NOT_USE，Filter 还实现了一个属性页。

最后，我们要来定义两个导出函数：DllRegisterServer 和 DllUnregisterServer，代码如下：

```
STDAPI DllRegisterServer()
{
    return AMovieDllRegisterServer2( TRUE );
}
STDAPI DllUnregisterServer()
{
    return AMovieDllRegisterServer2( FALSE );
}
```

让 Filter 项目编译成功后自动注册？

打开 VC 的菜单，选择 Project | Settings | Custom Build，在这一页的各项中输入如下代码即可：

```
Description: Register filter
Commands: regsvr32 /s /c "$(TargetPath)"
          echo regsvr32 exec. time > "$(TargetDir)\$(TargetName).trg"
Outputs: $(TargetDir)\$(TargetName).trg
```

4.5.2 框架函数的实现

CreateInstance 函数的实现

实现 CreateInstance 函数可以创建 Filter 对象实例，代码如下：

```
CUnknown * WINAPI CFilterTitleOverlay::CreateInstance(LPUNKNOWN punk,
HRESULT *phr)
{
    CFilterTitleOverlay *pNewObject = new
CFilterTitleOverlay(NAME("TitleOverlay"), punk, phr);
    if (pNewObject == NULL)
    {
        *phr = E_OUTOFMEMORY;
    }
    return pNewObject;
}
```

CheckInputType 函数的实现

实现 CheckInputType 函数可以完成输入 Pin 上的媒体类型检查，代码如下：

```
// 我们需要支持 RGB32, RGB24, RGB565, RGB555 等格式
HRESULT CFilterTitleOverlay::CheckInputType(const CMediaType* mtIn)
{
    // Dynamic format change will never be allowed!
    if (IsStopped() && *mtIn->Type() == MEDIATYPE_Video)
```

```
    {
        if (*mtIn->Subtype() == MEDIASUBTYPE_RGB32 ||
            *mtIn->Subtype() == MEDIASUBTYPE_RGB24 ||
            *mtIn->Subtype() == MEDIASUBTYPE_RGB555 ||
            *mtIn->Subtype() == MEDIASUBTYPE_RGB565)
        {
            return NOERROR;
        }
    }
    return E_INVALIDARG;
}
```

Transform 函数的实现

实现 Transform 函数可以完成字符叠加函数的调用，代码如下：

```
HRESULT CFilterTitleOverlay::Transform(IMediaSample *pSample)
{
    // 如果我们不能从输入 Pin 的媒体类型中获得帧率参数信息，
    // 则根据第一个 Sample 上的时间戳来计算
    if (mNeedEstimateFrameRate)
    {
        mNeedEstimateFrameRate = FALSE;
        REFERENCE_TIME  startTime = 0;
        REFERENCE_TIME  endTime   = 0;
        double          estimated = 25;
        if (SUCCEEDED(pSample->GetTime(&startTime, &endTime)))
        {
            estimated = 1.0 * UNITS / (endTime - startTime);
        }
        mOverlayController->SetEstimatedFrameRate(estimated);
    }
    if (mOverlayType != OT_NONE)
    {
        PBYTE  pData = NULL;
        pSample->GetPointer(&pData);
        mOverlayController->DoTitleOverlay(pData);
    }
    return NOERROR;
}
```

我们看到，mOverlayController->DoTitleOverlay(pData)是真正实现字符叠加的函数调用。

NonDelegatingQueryInterface 函数的实现

实现 NonDelegatingQueryInterface 函数可以“暴露”Filter 支持的接口，代码如下：

```
// Basic COM - used here to reveal our own interfaces
STDMETHODIMP CFilterTitleOverlay::NonDelegatingQueryInterface(REFIID
riid, void ** ppv)
{
```

```
        CheckPointer(ppv, E_POINTER);
        if (riid == IID_ISpecifyPropertyPages)     // 属性页接口
        {
                return GetInterface((ISpecifyPropertyPages *) this, ppv);
        }
        else if (riid == IID_ITitleOverlay)        // 自定义接口
        {
                return GetInterface((ITitleOverlay *) this, ppv);
        }
        else
        {
                return   CTransInPlaceFilter::NonDelegatingQueryInterface(riid,
ppv);
        }
    }
```

提示

需要在 CFilterTitleOverlay 类定义中加入 DECLARE_IUNKNOWN 声明。

CompleteConnect 函数的实现

实现 CompleteConnect 函数，可以在输入 Pin 成功连接后取得媒体类型描述，代码如下：

```
    HRESULT CFilterTitleOverlay::CompleteConnect(PIN_DIRECTION direction,
IPin *pReceivePin)
    {
        HRESULT hr = CTransInPlaceFilter::CompleteConnect(direction,
    pReceivePin);
        if (SUCCEEDED(hr) && direction == PINDIR_INPUT)
        {
                hr = SetInputVideoInfoToController();
        }
        return hr;
    }
```

其中，SetInputVideoInfoToController 函数将输入数据的格式设置给应用逻辑控制对象，该函数实现代码如下：

```
    HRESULT CFilterTitleOverlay::SetInputVideoInfoToController(void)
    {
        if (mOverlayController && m_pInput && m_pInput->IsConnected())
        {
                CMediaType  mt = m_pInput->CurrentMediaType();
                if (mt.formattype != FORMAT_VideoInfo)
                {
                        return E_FAIL;
                }
                RGB_FORMAT   colorSpace = FT_NONE;
                if (mt.subtype == MEDIASUBTYPE_RGB32)  // Determine RGB format
                {
```

```
                colorSpace = FT_RGB32;
            }
            else if (mt.subtype == MEDIASUBTYPE_RGB24)
            {
                colorSpace = FT_RGB24;
            }
            else if (mt.subtype == MEDIASUBTYPE_RGB555)
            {
                colorSpace = FT_RGB555;
            }
            else if (mt.subtype == MEDIASUBTYPE_RGB565)
            {
                colorSpace = FT_RGB565;
            }
            else if (mt.subtype == MEDIASUBTYPE_RGB8)
            {
                colorSpace = FT_RGB8;
            }

            mOverlayController->SetInputColorSpace(colorSpace);
            VIDEOINFOHEADER * pHeader = (VIDEOINFOHEADER *) mt.pbFormat;
            mNeedEstimateFrameRate = pHeader->AvgTimePerFrame>0? FALSE :
        TRUE;
            mOverlayController->SetInputVideoInfo(pHeader);
            return NOERROR;
    }
    return E_FAIL;
}
```

StartStreaming 和 StopStreaming 函数的实现

实现 StartStreaming 和 StopStreaming 函数，可以分别调用应用逻辑控制对象的相应函数，进行初始化和反初始化，代码如下：

```
HRESULT CFilterTitleOverlay::StartStreaming()
{
    BOOL pass = mOverlayController->StartTitleOverlay();
    return pass ? S_OK : E_FAIL;
}
HRESULT CFilterTitleOverlay::StopStreaming()
{
    mOverlayController->StopTitleOverlay();
    return NOERROR;
}
```

4.5.3　逻辑控制类的实现

逻辑控制类才是字符叠加应该真正实现的地方。下面首先介绍字符叠加原理。

一般输入的每个视频 Sample（非压缩格式）都带有一帧图像数据，字符叠加实际上就是将指定位置的图像像素值替换为字符图像的像素值。那么，怎么替换效率更高一点呢？一种直观的想法是，将图像帧选入 GDI 的 DC 中，再使用 GDI 函数 TextOut 和 DrawText 等

直接在图像帧上输出字符，然后再取出图像帧并返回给 Filter 框架。实际上，这种做法的效率不是很理想，要想做到实时叠加难度很大。

我们采用的一种方法是，在内存中首先建立一个二色位图，然后在这个位图上画出字符内容。于是，我们就得到了字符内容的一块点阵信息：比如 0 表示背景，1 表示前景字符。在实际叠加的时候，我们将图像帧指定位置的像素与字符点阵的像素位对应，如果字符点阵的像素位值为 0，则保持图像帧对应的像素值不变；如果为 1，则将图像帧对应的像素值替换为用户设置的字符颜色值。

创建字符内容的二色位图的代码如下：

```
BOOL COverlayController::CreateTitleDIBBits(void)
{
    // 创建一个与桌面兼容的内存 DC
    HDC hdc = CreateCompatibleDC(NULL);
    if (hdc)
    {
            // 创建字符位图
            HBITMAP hbm = ActualCreateTitleDIB(hdc);
            DeleteDC(hdc);
            if (hbm)
            {
                    DeleteObject(hbm);
                    return TRUE;
            }
    }
    return FALSE;
}
HBITMAP COverlayController::ActualCreateTitleDIB(HDC inDC)
{
    // 创建字符位图使用的 DIB 信息
    // 创建的位图初始为全黑背景，而在有输出的区域为白底黑字
    struct {
            BITMAPINFOHEADER bmiHeader;
            DWORD rgbEntries[2];
    } bmi =
    {
            {
                    sizeof(BITMAPINFOHEADER),
                    0,
                    0,
                    1,
                    1,
                    BI_RGB,
                    0,
                    0,
                    0
            },
            {
                    0x00000000,
```

```
                    0xFFFFFFFF
            }
    };
    // 设置字体
    CAutoFont  autoFont;
    if (mIsFontChanged)
    {
            autoFont.CreateFont(mTitleFont);
            autoFont.SelectToDC(inDC);
    }
    // 得到整个字符内容的宽和高，以决定我们将要创建的位图的大小
    GetTextExtentPoint32(inDC, mTitle, lstrlen(mTitle), &mTitleSize);
// 给子类一个机会改变将要创建的位图的大小
    if (!ValidateTitleDIBSize())
    {
            return NULL;
    }
    // 设置将要创建的位图的大小
    bmi.bmiHeader.biHeight = mTitleSize.cy;
    bmi.bmiHeader.biWidth  = mTitleSize.cx;
    HBITMAP hbm = CreateDIBitmap(inDC, &bmi.bmiHeader, 0, NULL, NULL, 0);
    BOOL   pass = (hbm != NULL);
    if (pass)
    {
            // 在位图上输出字符内容
            HGDIOBJ hobj = SelectObject(inDC, hbm);
            pass = ExtTextOut(inDC, 0, 0, ETO_OPAQUE | ETO_CLIPPED, NULL,
                    mTitle, lstrlen(mTitle), NULL);
            SelectObject(inDC, hobj);
    }
// 获取位图的数据（相当于点阵信息）
    if (pass)
    {
            ReleaseTitleDIB();
            // Attention: To get bitmap data from the DIB object,
            // the scan line must be a multiple of 4 (DWORD)!
            // If the actual bitmap data is not exactly fit for DWORD,
            // The rest of DWORD bits will be filled automatically.
            // So we should expand to bytes and round up to a multiple of 4.
            mDIBWidthInBytes = ((mTitleSize.cx + 31) >> 3) & ~3;

            mTitleDIBBits    = new BYTE[mDIBWidthInBytes * mTitleSize.cy];
            memset(mTitleDIBBits, 0, mDIBWidthInBytes * mTitleSize.cy);
            LONG lLines = GetDIBits(inDC, hbm, 0, mTitleSize.cy,
(PVOID)mTitleDIBBits, (BITMAPINFO *)&bmi, DIB_RGB_COLORS);
            pass = (lLines != 0);
    }
    if (!pass && hbm)
    {
            DeleteObject(hbm);
```

```
        hbm = NULL;
    }
    return hbm;
}
```

在实际的字符叠加过程中，还有一个问题，就是在设计逻辑控制类的时候，我们事先并不能知道输入的图像到底是 RGB 的哪种格式。这需要在 Filter 的输入 Pin 连接完成后设置进来。然而，不同的 RGB 格式，每个像素使用的字节数不一样，像素值的替换方式也就不一样。我们希望用一种统一的方式来控制。于是，我们对像素进行了抽象，设计了如图 4.30 所示的一个像素操作类结构。

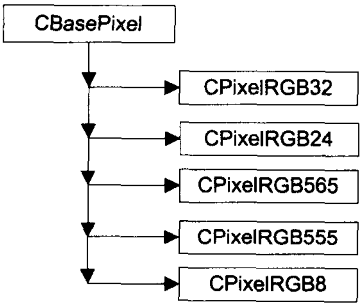

图 4.30　像素操作类结构

CBasePixel 类的定义代码如下：

```
class CBasePixel
{
protected:
    unsigned char    m_TargetR;       // 字符颜色的 RGB 值
    unsigned char    m_TargetG;
    unsigned char    m_TargetB;
    unsigned char    m_PixelSize;     // 每个像素使用的字节数

public:
    CBasePixel();
    ~CBasePixel();
public:
    void SetTargetColor(unsigned char inR, unsigned char inG, unsigned
char inB);
    void SetPixelSize(int inSize);
    int  GetPixelSize(void) {return m_PixelSize;}
    // 直接替换的叠加方式
    virtual void ConvertByCover(unsigned char * inPixel);
    // 使用反色叠加，使前景的字符在背景视频图像上更突出
    virtual void ConvertByReverse(unsigned char * inPixel);
    // 指向下一个像素
    unsigned char * NextPixel(unsigned char * inCurrent);
    // 指向下 N 个像素
    unsigned char * NextNPixel(unsigned char * inCurrent, int inCount);
protected:
```

```
    virtual void SideEffectColorChanged(void);
};
```

CBasePixel 的子类分别实现各自的 ConvertByCover 函数，用于实际像素值的替换。我们看到控制类 COverlayController 定义了一个 CBasePixel 类型的成员对象指针。根据输入图像的具体格式，我们生成不同的 CBasePixel 子类对象实例。于是，我们可以统一调用 CBasePixel::ConvertByCover 或 CBasePixel::ConvertByReverse 来完成像素值的替换。

我们再回到控制类的实现问题上。我们看到，控制类上定义了很多成员函数，用于参数的设置。另外定义了 3 个重要的公共函数：

```
virtual BOOL StartTitleOverlay(void);// Filter 进入运行状态时调用，以便初始化
virtual BOOL StopTitleOverlay(void);  // Filter 进入停止状态时调用，以便反初始化
BOOL DoTitleOverlay(PBYTE inImage);   // 真正的字符叠加函数
```

DoTitleOverlay 的实现代码如下：

```
BOOL COverlayController::DoTitleOverlay(PBYTE inImage)
{
    if (!mCanDoOverlay)
    {
            return FALSE;
    }

    BOOL pass = BeforeActualOverlay();      // 开始叠加前进行必要的准备
    if (pass)
    {
            pass = ActualOverlay(inImage); // 真正的叠加
    }
    if (pass)
    {
            pass = AfterActualOverlay();   // 叠加后的“善后”工作
    }
    return pass;
}
```

其中，BeforeActualOverlay、ActualOverlay、AfterActualOverlay 都是虚函数（子类都可以重新实现）。之所以如此编写，是因为 COverlayController 是一个基类，更多的职责在于设计应用框架，而留出虚函数让子类根据具体应用去定制。

ActualOverlay 函数的实现代码如下：

```
BOOL COverlayController::ActualOverlay(PBYTE inImage)
{
    // Now copy the data from the DIB section (which is usually bottom-up)
    // but first check if it's too big
    if (mImageWidth > mStartPos.x && mImageHeight > mStartPos.y &&
            mTitleSize.cx > 0 && mTitleSize.cy > 0)
    {
            long actualOverlayWidth    = min((mImageWidth  - mStartPos.x),
mTitleSize.cx);
            long actualOverlayHeight  = min((mImageHeight  - mStartPos.y),
```

```
mTitleSize.cy);
            // Image may be bottom-up, may be top-down
            // Anyway retrieve the pointer which point to the top line
            PBYTE   pTopLine       = NULL;
            long    strideInBytes = 0;
            if (mIsBottomUpImage)
            {
                   strideInBytes = -mImageWidthInBytes;
                    pTopLine = inImage + mImageWidthInBytes * (mImage-
                               Height - 1);
            }
            else
            {
                   strideInBytes = mImageWidthInBytes;
                   pTopLine       = inImage;
            }

            PBYTE  pStartPos = pTopLine + mStartPos.y * strideInBytes
            pStartPos += (mStartPos.x * mImageBitCount / 8);
            for (DWORD dwY = 0; dwY < (DWORD)actualOverlayHeight; dwY++)
            {
                   PBYTE pbTitle   = mTitleDIBBits;
                   pbTitle  +=  (mDIBWidthInBytes  *  ((DWORD)mTitleSize.cy  -
dwY - 1));
                   for(DWORD dwX =0;dwX < (DWORD)actualOverlayWidth; dwX++)
                   {
                    // dwX & 7, value from 0 - 7
                    // 0x80 >> (dwX & 7), value from 10000000 to 00000001
                    // dwX >> 3, value add one every eight
                    // If the source bit is 0, the background. If 1, draw
                   // the text.
                    if ( !((0x80 >> (dwX & 7)) & pbTitle[dwX >> 3]) )
                    {
                           PBYTE pbPixel = mPixelConverter->
                                           NextNPixel(pStartPos,dwX);
                                if (mIsOverlayByCover)
                                {
                                          mPixelConverter
                                          ->ConvertByCover(pbPixel);
                                }
                                else
                                {
                                          mPixelConverter
                                          ->ConvertByReverse(pbPixel);
                                }
                          }
                   }
                   pStartPos += strideInBytes;
            }
            return TRUE;
```

```
    }
    return FALSE;
}
```

上述函数中，首先判断叠加位置的有效性；然后计算得到一个指向图像帧第一行数据的指针。图像一般有两种扫描方式：从下往上（Bottom-Up）和从上往下（Top-Down）。前者多用于 GDI 环境，图像数据存储时，最后一行的数据放在最前面，然后依次放倒数第 2 行的数据……这时候计算指向图像第 1 行数据的指针，我们必须从图像数据开始地址计算一个偏移量；后者多用于 DirectDraw 环境，是正常的图像数据存储格式。图像的扫描方式用 BITMAPINFOHEADER 数据结构的 biHeight 成员值的正负来表示：正数表示从下往上，负数表示从上往下。mIsBottomUpImage 的取值如下：

```
mIsBottomUpImage = inInfoHeader->bmiHeader.biHeight > 0 ? TRUE : FALSE;
```

再下来就是一个两层循环，完成了字符内容像素值在图像帧中对应位置的替换。

关于逻辑控制类的扩展问题，比如要叠加系统时间，我们就可以从 COverlayController 中派生一个类 CSysTimeOverlayController。因为系统时间是一直在变的，所以这个控制类的特点是，每次叠加前都必须重新生成字符内容的点阵信息。我们主要通过重新实现 BeforeActualOverlay 函数来完成，代码如下：

```
BOOL CSysTimeOverlayController::BeforeActualOverlay(void)
{
    // 更新当前系统时间
    SYSTEMTIME    systemTime, localTime;
    GetSystemTime(&systemTime); // This is Coordinated Universal Time (UTC)
    SystemTimeToTzSpecificLocalTime(NULL, &systemTime, &localTime);
    sprintf(mTitle, "%4d-%02d-%02d(%02d:%02d:%02d)", localTime.wYear,
localTime.wMonth,
            localTime.wDay, localTime.wHour, localTime.wMinute,
localTime.wSecond);
// 重新创建字符点阵信息
    BOOL pass = CreateTitleDIBBits();
    return pass;
}
```

再比如我们要扩展滚动字幕叠加控制，同样也是从 COverlayController 中派生一个类 CScrollController。因为滚动字幕的叠加坐标以及字符内容的显示区域需要实时计算获得，这个控制类的主要实现是，重写 BeforeActualOverlay 函数以便每次叠加前完成上述计算工作，并且重写 ActualOverlay 函数，完成在动态指定的坐标上叠加有效区域的字符内容。代码如下：

```
BOOL CScrollController::BeforeActualOverlay(void)
{
    BOOL pass = COverlayController::BeforeActualOverlay();
    if (pass)
    {
          // Calculate for the next progress
          long actualProgress = mOverlayCounter - mOverlayStartTime;
```

```
            mStartPos.x = long(mImageWidth - 1 - mScrollStride *
actualProgress);
            if (mStartPos.x < 0)
            {
                    // Moving out of the left side
                    mValidTitleRect.left = -mStartPos.x;
                    if (mValidTitleRect.left >= mTitleSize.cx)
                    {
                            mValidTitleRect.left = mTitleSize.cx - 1;
                    }
                    mStartPos.x = 0;
                    if (mTitleSize.cx - mValidTitleRect.left <= mImageWidth)
                    {
                            mValidTitleRect.right = mTitleSize.cx - 1;
                    }
                    else
                    {
                            mValidTitleRect.right = mValidTitleRect.left +
mImageWidth;
                    }
            }
            else
            {
                    // Moving in the image-width range
                    mValidTitleRect.left = 0;
                    long currentLength = mImageWidth - mStartPos.x;
                    if (currentLength >= mTitleSize.cx)
                    {
                            mValidTitleRect.right = mTitleSize.cx - 1;
                    }
                    else
                    {
                            mValidTitleRect.right = currentLength;
                    }
            }
            pass = (mValidTitleRect.right > mValidTitleRect.left);
    }
    return pass;
}
// mStartPos 为每次叠加前计算所得的开始坐标
// mValidTitleRect 为有效的字符区域
BOOL CScrollController::ActualOverlay(PBYTE inImage)
{
    if (mImageHeight > mTitleSize.cy && mTitleSize.cx > 0 &&
mTitleSize.cy > 0)
    {
            // Image may be bottom-up, may be top-down
            // Anyway retrieve the pointer which point to the top line
            PBYTE   pTopLine        = NULL;
            long    strideInBytes = 0;
```

```
            if (mIsBottomUpImage)
            {
                    strideInBytes = -mImageWidthInBytes;
                pTopLine = inImage + mImageWidthInBytes * (mImage-
                        Height - 1);
            }
            else
            {
                    strideInBytes = mImageWidthInBytes;
                    pTopLine = inImage;
            }
            PBYTE  pStartPos = pTopLine + mStartPos.y * strideInBytes;
            pStartPos += (mStartPos.x * mImageBitCount / 8);
            for (DWORD dwY = 0; dwY < (DWORD)mTitleSize.cy; dwY++)
            {
                    PBYTE pbTitle = mTitleDIBBits;
                    pbTitle  +=  (mDIBWidthInBytes  *  ((DWORD)mTitleSize.cy  -
dwY - 1));
                    // Point to the valid start position of title DIB
                    pbTitle += (mValidTitleRect.left >> 3);
                    long  startLeft = mValidTitleRect.left % 8;
                    long  endRight  = startLeft + mValidTitleRect.right;
                    endRight -= mValidTitleRect.left;
                    for (long dwX = startLeft; dwX < endRight; dwX++)
                    {
                            if ( !((0x80 >> (dwX & 7)) & pbTitle[dwX >> 3]) )
                            {
                                    PBYTE pbPixel = mPixelConverter-
    >NextNPixel(pStartPos, dwX - startLeft);
                                    if (mIsOverlayByCover)
                                    {
                                            mPixelConverter-
>ConvertByCover(pbPixel);
                                    }
                                    else
                                    {
                                            mPixelConverter-
>ConvertByReverse(pbPixel);
                                    }
                            }
                    }
                    pStartPos += strideInBytes;
            }
        }
        return TRUE;
    }
```

4.5.4　自定义接口的实现

自定义接口的实现其实很简单。首先定义接口方法，代码如下：

```
// ITitleOverlay.h
// Desc: DirectShow sample code - custom interface

#ifndef __H_ITitleOverlay__
#define __H_ITitleOverlay__

#ifdef __cplusplus
extern "C" {
#endif

// ITitleOverlay's GUID

// {5E5B3386-6F9F-4a47-AC8B-7A302138D7FF}
DEFINE_GUID(IID_ITitleOverlay,
0x5e5b3386, 0x6f9f, 0x4a47, 0xac, 0x8b, 0x7a, 0x30, 0x21, 0x38, 0xd7,
0xff);
// ITitleOverlay
DECLARE_INTERFACE_(ITitleOverlay, IUnknown)
{
    // 设置 Filter 进行叠加的类型，如果需要改变类型，必须第 1 个设置这个函数，
    // 调用这个函数成功后，才能调用其他的函数进行参数设置
    // 可以设置的叠加类型参见枚举类型 OVERLAY_TYPE 的定义
    STDMETHOD(put_TitleOverlayType) (THIS_
            long inOverlayType
    ) PURE;
    STDMETHOD(get_TitleOverlayType) (THIS_
            long * outOverlayType
    ) PURE;
    // 设置像素转换的方式（可选），默认使用直接替换方式
    // 当字符颜色与视频背景颜色接近时，使用翻转叠加方式比较合适
    STDMETHOD(put_TitleOverlayStyle) (THIS_
            int inUsingCover
    ) PURE;
    STDMETHOD(get_TitleOverlayStyle) (THIS_
            int * outUsingCover
    ) PURE;
    // 设置欲叠加字符的内容
    STDMETHOD(put_Title) (THIS_
            const char * inTitle, int inLength
    ) PURE;
    STDMETHOD(get_Title) (THIS_
            char * outBuffer, int * outLength
    ) PURE;
    // 设置欲叠加字符的颜色，分别为 R, G, B 分量值
    STDMETHOD(put_TitleColor) (THIS_
            BYTE inR, BYTE inG, BYTE inB
    ) PURE;
    STDMETHOD(get_TitleColor) (THIS_
            BYTE * outR, BYTE * outG, BYTE * outB
    ) PURE;
```

```
    // 设置欲叠加字符的开始坐标
    STDMETHOD(put_TitleStartPosition) (THIS_
          POINT inStartPos
    ) PURE;
    STDMETHOD(get_TitleStartPosition) (THIS_
          POINT * outStartPos
    ) PURE;
    // 设置欲叠加字符的字体，参数采用 Windows GDI 的 LOGFONT 数据结构
    STDMETHOD(put_TitleFont) (THIS_
          LOGFONT inFont
    ) PURE;
    STDMETHOD(get_TitleFont) (THIS_
          LOGFONT * outFont
    ) PURE;
    // 设置欲叠加字符的生存周期，一个参数是开始时间，一个参数是结束时间，以 s 为单位
    // 如果 inEnd 值为-1，表示一直迭加到视频结束
    STDMETHOD(put_TitleDuration) (THIS_
          double inStart, double inEnd
    ) PURE;
    STDMETHOD(get_TitleDuration) (THIS_
          double * outStart, double * outEnd
    ) PURE;
};

#ifdef __cplusplus
}
#endif

#endif // __H_ITitleOverlay__
```

然后将 Filter 类从 ITitleOverlay 派生，并声明 ITitleOverlay 的所有接口方法，代码如下：

```
// 从 ITitleOverlay 中继承 Filter 类
class CFilterTitleOverlay : public CTransInPlaceFilter
                         , public ITitleOverlay
{
      // ...

      // --- ITitleOverlay methods ---
      STDMETHODIMP put_TitleOverlayType(long inOverlayType);
      STDMETHODIMP get_TitleOverlayType(long * outOverlayType);
      STDMETHODIMP put_TitleOverlayStyle(int inUsingCover);
      STDMETHODIMP get_TitleOverlayStyle(int * outUsingCover);
      STDMETHODIMP put_Title(const char * inTitle, int inLength);
      STDMETHODIMP get_Title(char * outBuffer, int * outLength);
      STDMETHODIMP put_TitleColor(BYTE inR, BYTE inG, BYTE inB);
      STDMETHODIMP get_TitleColor(BYTE * outR, BYTE * outG, BYTE * outB);
      STDMETHODIMP put_TitleStartPosition(POINT inStartPos);
      STDMETHODIMP get_TitleStartPosition(POINT * outStartPos);
      STDMETHODIMP put_TitleFont(LOGFONT inFont);
```

```
        STDMETHODIMP get_TitleFont(LOGFONT * outFont);
        STDMETHODIMP put_TitleDuration(double inStart, double inEnd);
        STDMETHODIMP get_TitleDuration(double * outStart, double * outEnd);
}
```

在 Filter 类的实现文件中实现这些接口方法（其实大都是调用应用逻辑控制类的相应函数），代码如下：

```
// --- ITitleOverlay methods ---
STDMETHODIMP CFilterTitleOverlay::put_TitleOverlayType(long
inOverlayType)
{
    CAutoLock   lockit(&mITitleOverlaySync);
    // When filter graph is active, change overlay type is not allowed!
    if (m_State != State_Stopped)
    {
            return E_UNEXPECTED;
    }
    if (mOverlayType != (OVERLAY_TYPE)inOverlayType)
    {
            mOverlayType = (OVERLAY_TYPE)inOverlayType;
            SideEffectOverlayTypeChanged();
    }
    return NOERROR;
}
// 当字符叠加的类型改变时，需要生成新的应用逻辑控制类对象实例，
// 并且在新生成的控制对象上设置参数信息
void CFilterTitleOverlay::SideEffectOverlayTypeChanged(void)
{
    ReleaseOverlayController();
    switch (mOverlayType)
    {
    case OT_SYSTIME:
            mOverlayController = new CSysTimeOverlayController();
            break;

    case OT_SCROLL_TOP:
            mOverlayController = new CScrollController();
            ((CScrollController*) mOverlayController)->
SetScrollBottomOrTop(FALSE);
            break;
    case OT_SCROLL_BOTTOM:
            mOverlayController = new CScrollController();
            ((CScrollController*) mOverlayController)->
SetScrollBottomOrTop(TRUE);
            break;
    case OT_NONE:
    case OT_STATIC:
    default:
            mOverlayController = new COverlayController();
    }
```

```
        // Make sure to set input video info to the new controller
        SetInputVideoInfoToController();
    }

    STDMETHODIMP CFilterTitleOverlay::get_TitleOverlayType(long *
outOverlayType)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        *outOverlayType = mOverlayType;
        return NOERROR;
    }
    STDMETHODIMP CFilterTitleOverlay::put_TitleOverlayStyle(int inUsingCover)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->SetOverlayStyle(inUsingCover);
        return NOERROR;
    }
    STDMETHODIMP CFilterTitleOverlay::get_TitleOverlayStyle(int *
outUsingCover)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->GetOverlayStyle(outUsingCover);
        return NOERROR;
    }
    STDMETHODIMP  CFilterTitleOverlay::put_Title(const  char  *  inTitle,  int
inLength)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        // When filter graph is active, change title text is not allowed!
        if (m_State != State_Stopped)
        {
               return E_UNEXPECTED;
        }
        mOverlayController->SetTitle(inTitle, inLength);
        return NOERROR;
    }
    STDMETHODIMP CFilterTitleOverlay::get_Title(char * outBuffer, int *
outLength)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        *outLength = mOverlayController->GetTitle(outBuffer);
        return NOERROR;
    }
    STDMETHODIMP  CFilterTitleOverlay::put_TitleColor(BYTE  inR,  BYTE  inG,
BYTE inB)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->SetTitleColor(inR, inG, inB);
        return NOERROR;
    }
```

```
    STDMETHODIMP  CFilterTitleOverlay::get_TitleColor(BYTE  *  outR,  BYTE  *
outG, BYTE * outB)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->GetTitleColor(outR, outG, outB);
        return NOERROR;
    }
    STDMETHODIMP CFilterTitleOverlay::put_TitleStartPosition(POINT
    inStartPos)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->SetTitleStartPosition(inStartPos);
        return NOERROR;
    }
    STDMETHODIMP CFilterTitleOverlay::get_TitleStartPosition(POINT *
outStartPos)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->GetTitleStartPosition(outStartPos);
        return NOERROR;
    }
    STDMETHODIMP CFilterTitleOverlay::put_TitleFont(LOGFONT inFont)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->SetTitleFont(inFont);
        return NOERROR;
    }
    STDMETHODIMP CFilterTitleOverlay::get_TitleFont(LOGFONT * outFont)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->GetTitleFont(outFont);
        return NOERROR;
    }
    STDMETHODIMP   CFilterTitleOverlay::put_TitleDuration(double   inStart,
double inEnd)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->SetTitleDuration(inStart, inEnd);
        return NOERROR;
    }
    STDMETHODIMP  CFilterTitleOverlay::get_TitleDuration(double  *  outStart,
double * outEnd)
    {
        CAutoLock   lockit(&mITitleOverlaySync);
        mOverlayController->GetTitleDuration(outStart, outEnd);
        return NOERROR;
    }
```

最后，在 NonDelegatingQueryInterface 函数中“暴露”我们的 ITitleOverlay 接口，代码如下：

```
STDMETHODIMP CFilterTitleOverlay::NonDelegatingQueryInterface(REFIID
riid, void ** ppv)
{
    CheckPointer(ppv, E_POINTER);
    if (riid == IID_ITitleOverlay)
    {
            return GetInterface((ITitleOverlay *) this, ppv);
    }
    else
    {
            return CTransInPlaceFilter::NonDelegatingQueryInterface(riid,
ppv);
    }
}
```

4.5.5　属性页的实现

属性页是用户与 Filter 交互的简单界面（一般在商业软件中不推荐将 Filter 的属性页直接显示给用户）。Filter 属性页更多地用于开发时候的接口调试。当然，Filter 也可以不实现属性页。

在“字符叠加 Filter”中，我们需要实现一个属性页。首先使用 VC 的资源编辑器生成一个对话框以及一个标识对话框标题的字符串 ID，并编辑对话框上的界面元素，包括选择字符叠加效果的组合框、字符内容输入框、字符叠加的位置输入框、字体选择按钮等，如图 4.31 所示。

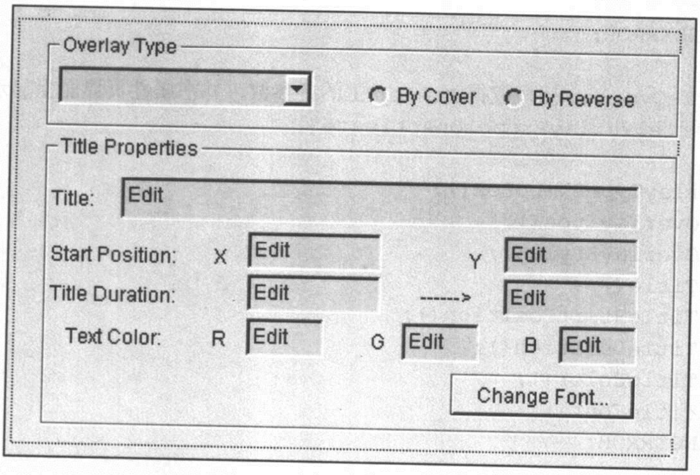

图 4.31　Filter 属性页界面

然后从 CBasePropertyPage 类中派生一个子类 CTitleOverlayProp，用于实现我们的属性页。在 CTitleOverlayProp 类实现中，我们主要重写了如下函数：

```
// 创建属性页 COM 对象实例
CUnknown * WINAPI CTitleOverlayProp::CreateInstance(LPUNKNOWN lpunk,
HRESULT *phr)
{
    CUnknown *punk = new CTitleOverlayProp(lpunk, phr);
```

```
    if (punk == NULL)
    {
        *phr = E_OUTOFMEMORY;
    }
    return punk;
}
// 获得 ITitleOverlay 接口
HRESULT CTitleOverlayProp::OnConnect(IUnknown *pUnknown)
{
    HRESULT hr = pUnknown->QueryInterface(IID_ITitleOverlay, (void **)
&mIOverlay);
    if (FAILED(hr))
    {
        return E_NOINTERFACE;
    }
    ASSERT(mIOverlay);
    return NOERROR;
}
// 释放 ITitleOverlay 接口
HRESULT CTitleOverlayProp::OnDisconnect()
{
    // Release of Interface
    if (mIOverlay == NULL)
        return E_UNEXPECTED;
    mIOverlay->Release();
    mIOverlay = NULL;
    return NOERROR;
}
// 通过 ITitleOverlay 接口取得 Filter 上的各参数，并在属性页界面上显示
HRESULT CTitleOverlayProp::OnActivate()
{
    FillOverlayTypeComboBox();
    ReflectOverlayType();
    ReflectOverlayStyle();
    ReflectTitle();
    ReflectTitleStartPosition();
    ReflectTitleDuration();
    ReflectTitleColor();
    ReflectTitleFont();
    return NOERROR;
}
// 将属性页界面上用户设置的各参数，通过 ITitleOverlay 接口设置给 Filter
HRESULT CTitleOverlayProp::OnApplyChanges()
{
    EnterOverlayType();  // This must be first invoked!!
    EnterOverlayStyle();
    EnterTitle();
    EnterTitleStartPosition();
    EnterTitleDuration();
    EnterTitleColor();
```

```
    EnterTitleFont();
    return NOERROR;
}
```

注意，上述 OnActivate 和 OnApplyChanges 函数实现中调用的各函数都是 Ctitle-Overlay Prop 实现的私有成员函数，用于操作各个参数的读取和设置。代码如下：

```
void CTitleOverlayProp::ReflectTitle(void)
{
    int  titleLength = 0;
    mIOverlay->get_Title(NULL, &titleLength);  // 获得字符长度
    if (titleLength > 0)
    {
          char * szTitle = new char[titleLength];
          mIOverlay->get_Title(szTitle, &titleLength);  // 获得字符内容
          SetWindowText(m_hEditTilte, szTitle);
          delete[] szTitle;
    }
}
void CTitleOverlayProp::EnterTitle(void)
{
    char szTitle[2000];
    // 设置字符内容
    if (GetWindowText(m_hEditTilte, szTitle, 2000))
    {
          mIOverlay->put_Title(szTitle, strlen(szTitle));
    }
    else
    {
          mIOverlay->put_Title("", 0);
    }
}
```

另外，我们还实现了 OnReceiveMessage 函数，用于处理属性页窗口的消息，代码如下：

```
BOOL CTitleOverlayProp::OnReceiveMessage(HWND hwnd,
                                         UINT uMsg,
                                         WPARAM wParam,
                                         LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_INITDIALOG:
        {
                // 得到各界面元素的窗口句柄
                m_hOverlayType = GetDlgItem(hwnd, IDC_COMBO_OVERLAY_TYPE);
                m_hEditTilte     = GetDlgItem(hwnd, IDC_EDIT_TITLE);
                m_hEditStartX    = GetDlgItem(hwnd, IDC_EDIT_STARTX);
                m_hEditStartY    = GetDlgItem(hwnd, IDC_EDIT_STARTY);
                m_hEditStartTime  = GetDlgItem(hwnd, IDC_EDIT_STARTTIME);
```

```
                m_hEditEndTime    = GetDlgItem(hwnd, IDC_EDIT_ENDTIME);
                m_hEditColorR     = GetDlgItem(hwnd, IDC_EDIT_COLORR);
                m_hEditColorG     = GetDlgItem(hwnd, IDC_EDIT_COLORG);
                m_hEditColorB     = GetDlgItem(hwnd, IDC_EDIT_COLORB);
                break;
        }

        case WM_COMMAND:
        {
                if (HIWORD(wParam) == BN_CLICKED)
                {
                        switch (LOWORD(wParam))
                        {
                                case IDC_BUTTON_CHANGE_FONT: // 改变字体
                                        OnButtonChangeFont();
                                        break;
                        }
                }
                SetDirty();
            break;
        }

    }
    return CBasePropertyPage::OnReceiveMessage(hwnd,uMsg,wParam,lParam);
}
```

其中，OnButtonChangeFont 函数实现为弹出 Windows 标准的字体选择框，代码如下：

```
void CTitleOverlayProp::OnButtonChangeFont(void)
{
    // Update the RGB values
    BYTE  colorR = 0, colorG = 0, colorB = 0;
    char  szColor[100];
    GetWindowText(m_hEditColorR, szColor, 100);
    colorR = atoi(szColor);
    GetWindowText(m_hEditColorG, szColor, 100);
    colorG = atoi(szColor);
    GetWindowText(m_hEditColorB, szColor, 100);
    colorB = atoi(szColor);
    mTitleColor = RGB(colorR, colorG, colorB);
    CHOOSEFONT    cf;
    // Initialize CHOOSEFONT
    ZeroMemory(&cf, sizeof(CHOOSEFONT));
    cf.lStructSize = sizeof(CHOOSEFONT);
    cf.hInstance    = g_hInst;
    cf.hwndOwner    = m_hwnd;
    cf.lpLogFont    = &mTitleFont;
    cf.rgbColors    = mTitleColor;
   cf.Flags         = CF_SCREENFONTS | CF_EFFECTS | CF_INITTOLOGFONTSTRUCT;
    // 调用标准的选择字体的 API 函数
```

```
        if (ChooseFont(&cf))
        {
                mIsFontChanged = TRUE;
                // Update the text color
                mTitleColor    = cf.rgbColors;
                ReflectTitleColor(GetRValue(mTitleColor),
GetGValue(mTitleColor), GetBValue(mTitleColor));
        }
    }
```

最后在 Filter 上实现 ISpecifyPropertyPages 的接口方法 GetPages，代码如下：

```
    STDMETHODIMP CFilterTitleOverlay::GetPages(CAUUID *pPages)
    {
        pPages->cElems = 1;
        pPages->pElems = (GUID *) CoTaskMemAlloc(sizeof(GUID));
        if (pPages->pElems == NULL)
        {
                return E_OUTOFMEMORY;
        }
        *(pPages->pElems) = CLSID_HQTitleOverlayProp;
        return NOERROR;
    }
```

并且在 NonDelegatingQueryInterface 函数中“暴露”ISpecifyPropertyPages 接口，代码如下：

```
    STDMETHODIMP CFilterTitleOverlay::NonDelegatingQueryInterface(REFIID
riid, void ** ppv)
    {
        CheckPointer(ppv, E_POINTER);

        if (riid == IID_ISpecifyPropertyPages)      // 支持属性页的接口
        {
                return GetInterface((ISpecifyPropertyPages *) this, ppv);
        }
        else if (riid == IID_ITitleOverlay)
        {
                return GetInterface((ITitleOverlay *) this, ppv);
        }
        else
        {
                return   CTransInPlaceFilter::NonDelegatingQueryInterface(riid,
ppv);
        }
    }
```

4.5.6 产权保护

如果我们开发的 Filter 具有一定的商业价值，我们当然不希望别人攫取我们的劳动成果。虽然技术保密不是我们开发人员的初衷，但我们也必须遵循市场的规律。这里，介绍

一种对 Filter 产权保护的简单实现方法，即在 Filter 创建的时候判断一下创建者是否“合法”——我们的 Filter 只允许被 GraphEdit 和自己的应用程序创建。

下面的代码可供参考：

```
CUnknown * WINAPI CFilterTitleOverlay::CreateInstance(LPUNKNOWN punk,
HRESULT *phr)
{
#if 1
    char    szCreatorPath[256], szCreatorName[256];
    ::strcpy(szCreatorPath, "");
    ::strcpy(szCreatorName, "");
    HMODULE hModule = ::GetModuleHandle(NULL);
    ::GetModuleFileName(hModule, szCreatorPath, 256);
    char * backSlash = ::strrchr(szCreatorPath, '\\');
    if (backSlash)
    {
          strcpy(szCreatorName, backSlash);
    }
    ::_strlwr(szCreatorName);
    // Please specify your app name with lowercase
    if (::strstr(szCreatorName, "graphedt") == NULL &&
    ::strstr(szCreatorName, "ourapp") == NULL)
    {
          *phr = E_FAIL;
          return NULL;
    }
#endif
    CFilterTitleOverlay *pNewObject = new
CFilterTitleOverlay(NAME("TitleOverlay"), punk, phr);
    if (pNewObject == NULL)
    {
          *phr = E_OUTOFMEMORY;
    }
    return pNewObject;
}
```

其中，ourapp 应该指定为我们的应用程序名字。

提示 实际上，上述方法并不能很好地保护 Filter。如果以二进制方式打开 Filter 的 AX 文件，可以查到 graphedt 和 ourapp 字符串。只需将其中一个字符串改成其他应用程序的名字，我们 Filter 的合法创建者就被篡改了。如果要实现更高一级的使用限制，我们应该考虑其他方法，比如实现一个设置/检查注册码的接口等。

4.6 Filter 的调试

Filter 的开发一般不可能是一举成功的。通常，我们需要经历一个测试、调试的过程。在这个过程中，我们会碰到一些问题，然后我们再想办法解决它们。其实，这也是一

个积累实践经验的过程，是一次培养自己解决实际问题能力的机会。

调试 Filter 使用的应用程序当然首选 DirectShow SDK 提供的 GraphEdit 工具软件（在 SDK 安装目录下的 Bin\DXUtils\graphedt.exe）。我们在 VC 中进行如下设置：选择菜单 Project | Settings | Debug，在这一页的 Executable for debug session 中选择 graphedt.exe 路径。如图 4.32 所示。

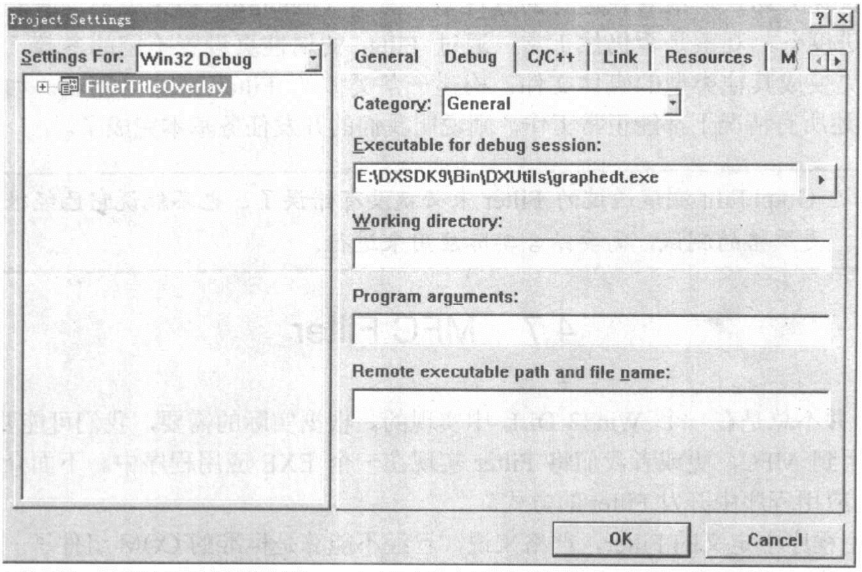

图 4.32　在 VC 中设置调试程序界面

当调试环境配置好后，我们就可以在 Filter 源文件中需要的地方设置断点，然后就是执行 VC 的调试命令。在运行起来的 GraphEdit 中构建如图 4.33 所示的一条 Filter 调试链路。

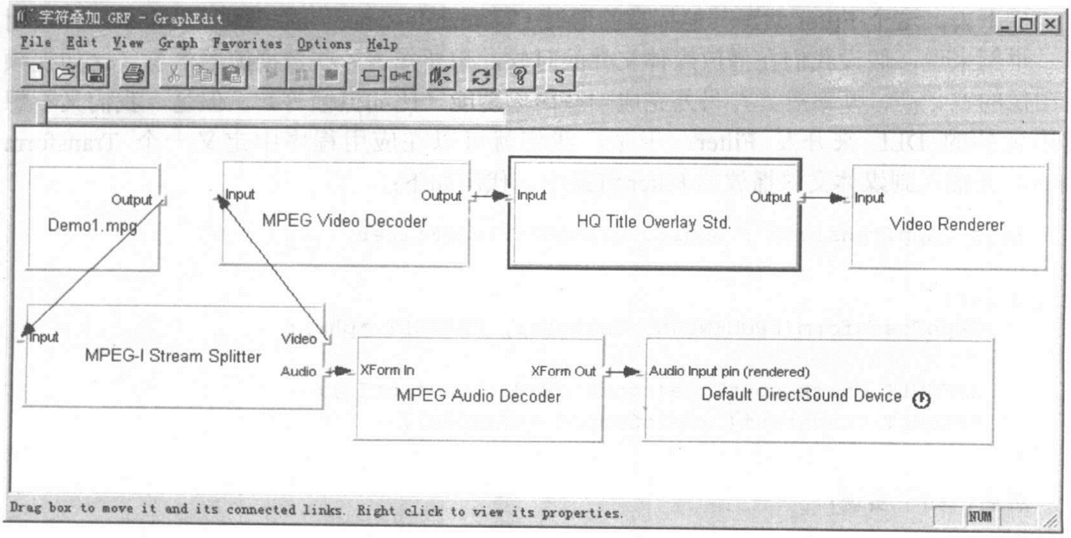

图 4.33　Filter 在 GraphEdit 中的调试链路

> **提示** 如果想要调试进入 Filter 项目，在 VC 中进行如下设置：选择 Project | Settings | Link，在这一页的 Category 中选择 General，然后确定将 Generate debug info 选中。

运行 Filter Graph，我们可以查看字符叠加是否正常。反复进行运行、暂停、停止，并无序（快速）切换，或者在运行的过程中突然将 GraphEdit 程序关掉，看看是否有问题。拖动进度条，观察是否仍然正常。通过 Filter 的属性页设置不同的参数，再进行测试。或者更换成其他类型的媒体文件，构建一条类似的 Filter 链路再测试。如果我们的 Filter 在上述所有情况下都能正常工作，则说明我们的开发任务基本完成了。

> **提示** 经过 GraphEdit 测试通过的 Filter 未必就没有错误了，也不能说它已经很“健壮”了。更严格的测试，需要结合实际应用来进行。

4.7 MFC Filter

Filter 并不总是在一个 Win32 DLL 中实现的。根据实际的需要，我们可能要在 Filter 的实现中用到 MFC，更或者我们将 Filter 实现在一个 EXE 应用程序中。下面介绍这样一种在 MFC 应用程序中开发 Filter 的方式。

在应用程序中定义的 Filter，严格来说，已经不能算是标准的 COM 组件了，它没有独立的实现模块，不需要注册，当然也不能通过标准的 COM 类工厂来创建。创建这种 Filter 的正确方法是，直接在应用程序中 new 出一个 Filter 类对象实例来。代码如下：

```
m_pFilterObject = new CFilterClass();
// Make the initial refcount 1 to match COM creation
m_pFilterObject ->AddRef();
```

接下去，这个 Filter 就跟其他标准的通过 CoCreateInstance 函数创建的 Filter 一样使用了。举例来说，假设我们在播放媒体文件的时候，想要知道一些数据流的信息（比如视频流图像的宽、高、帧率等），另外完成一些简单功能（比如抓帧等）。但是，我们又不想使用另外的 DLL 来开发 Filter。于是，我们就可以在应用程序中定义一个 Transform Filter，并插入到媒体文件播放的 Filter 链路中。代码如下：

```
class CAppTransform : public CTransInPlaceFilter
{
public:
    CAppTransform(LPUNKNOWN pUnkOuter, HRESULT *phr);

    HRESULT CheckInputType(const CMediaType* mtIn);
    HRESULT Transform(IMediaSample *pSample);
};
```

然后，如下实现：

```
CAppTransform::CAppTransform(LPUNKNOWN pUnkOuter, HRESULT *phr) :
CTransInPlaceFilter(NAME("App Trans-IP"), pUnkOuter, GUID_NULL, phr)
```

```
{
}
HRESULT CAppTransform::Transform(IMediaSample *pSample)
{
        // Override to do something inside the application
        // Such as grabbing a poster frame...
        // ...
    return S_OK;
}
// Check if we can support this specific proposed type and format
HRESULT CAppTransform::CheckInputType(const CMediaType *pmt)
{
    // We accept a series of raw media types
    if (pmt->majortype == MEDIATYPE_Video &&
          (pmt->subtype == MEDIASUBTYPE_RGB32 ||
          pmt->subtype == MEDIASUBTYPE_RGB24 ||
          pmt->subtype == MEDIASUBTYPE_RGB565 ||
          pmt->subtype == MEDIASUBTYPE_RGB555 ||
          pmt->subtype == MEDIASUBTYPE_UYVY ||
          pmt->subtype == MEDIASUBTYPE_YUY2))
    {
          return NOERROR;
    }
    return E_FAIL;
}
```

为了播放媒体文件，我们另外设计了一个类 CAppGraphBuilder，用于 Filter Graph 的构建。我们的思路是，首先使用 IGraphBuilder::RenderFile 构建一个完整的媒体文件回放链路，然后找到 Video Renderer（或者 Overlay Mixer），在它之前插入我们自己的 Transform Filter。代码如下：

```
// 给定一个媒体文件，创建一个回放链路，随后插入我们的 Transform Filter
HRESULT CAppGraphBuilder::BuildFromFile(LPCWSTR pszFile)
{
    DestroyGraph();
    // 创建一个 Filter Graph
    HRESULT hr = CoCreateInstance(
                CLSID_FilterGraph,
                NULL,
                CLSCTX_INPROC,
                IID_IGraphBuilder,
                (void**)&m_pGraph);
    if (FAILED(hr))
    {
          return hr;
    }
    AddToObjectTable();
    // 首先构建一个源文件的回放 Filter Graph
    hr = m_pGraph->RenderFile(pszFile, NULL);
    if (FAILED(hr))
```

```
    {
        return hr;
    }
    // 通过 IVideoWindow 接口找到 Video Renderer
    IBaseFilter* pVR;
    hr = FindFilterByInterface(IID_IVideoWindow, &pVR);
    if (FAILED(hr))
    {
        return hr;
    }
    // 注意，如果 Video Renderer 的上一级 Filter 是 Overlay Mixer，则将没有实际的
    // 数据传送到 Video Renderer 上，这种情况下，我们的 Filter 应该插在 Overlay
    // Mixer 之前
    IPin* pPin = InputPinOf(pVR);     // 取得 Video Renderer 的输入 Pin
    AM_MEDIA_TYPE mt;
    pPin->ConnectionMediaType(&mt);  // 取得连接用的媒体类型
    pPin->Release();
    CMediaType mtIn = mt;
    FreeMediaType(mt);
    if (mtIn.subtype == MEDIASUBTYPE_Overlay)
    {
        // This connection may be a overlay mixer
        // need to move upstream one place
        IBaseFilter* pOvMix = NULL;
        hr = NextUpstream(pVR, &pOvMix);
        pVR->Release();
        if (FAILED(hr))
        {
            return hr;
        }
        pVR = pOvMix;
    }
    // 创建我们的 Transform Filter
    CreateAppFilter();
    // 将我们的 Filter 插入到已有的 Filter 链路中
    hr = ConnectUpstreamOf(pVR, m_pFilter);
    pVR->Release();

    return hr;
}
// 创建我们的 Transform Filter，注意不是用 CoCreateInstance 函数
void CAppGraphBuilder::CreateAppFilter(void)
{
    if (m_pFilter)
    {
        m_pFilter->Release();
        m_pFilter = NULL;
    }
    HRESULT hr = S_OK;
    // 创建 Filter 对象实例
```

```
        m_pFilter = new CAppTransform(NULL, &hr);
        // 增加一个引用计数，防止 COM 的自动销毁
        m_pFilter->AddRef();
        // 获取 IBaseFilter 接口
        IBaseFilter* pFilter = NULL;
        hr = m_pFilter->QueryInterface(IID_IBaseFilter, (void**)&pFilter);
        if (SUCCEEDED(hr))
        {
                hr = m_pGraph->AddFilter(pFilter, L"App Transform");
                pFilter->Release();
        }
    }
    // 将 pTransform Filter（假设为一进一出）插入到 pFilter 的前面
    HRESULT CAppGraphBuilder::ConnectUpstreamOf(IBaseFilter* pFilter,
IBaseFilter* pTransform)
    {
        IPin* pPinIn = InputPinOf(pFilter);
        if (!pPinIn)
        {
                return E_FAIL;
        }
        // Get the peer output pin
        IPin* pPinOut = NULL;
        HRESULT hr = pPinIn->ConnectedTo(&pPinOut);
        if (FAILED(hr))
        {
                pPinIn->Release();
                return hr;
        }
        // Disconnect the current connection
        hr = m_pGraph->Disconnect(pPinOut);
        if (SUCCEEDED(hr))
        {
                hr = m_pGraph->Disconnect(pPinIn);
        }
        // Insert pTransform filter by connecting its input pin and output
        // pin
        if (SUCCEEDED(hr))
        {
                IPin* pPinInXfm = InputPinOf(pTransform);
                hr = m_pGraph->Connect(pPinOut, pPinInXfm);
                pPinInXfm->Release();
        }
        if (SUCCEEDED(hr))
        {
                IPin* pPinOutXfm = OutputPinOf(pTransform);
                hr = m_pGraph->Connect(pPinOutXfm, pPinIn);
                pPinOutXfm->Release();
        }
        pPinIn->Release();
```

```
    pPinOut->Release();
    return hr;
}
```

提示

更多的实现细节，请读者参考下载地址（见前言）中的 AppIPTransform 项目。

第 5 章　DirectShow 应用开发过程

5.1　开发环境的配置

大家已经知道，DirectShow 采用的是 COM 架构，而 DirectShow 应用程序也就是 COM 的客户程序。跟介绍 Filter 组件的开发过程一样，我们首先也来介绍一下 DirectShow 应用程序开发环境的配置。

5.1.1　需要包含的头文件

DirectShow SDK 建议，所有 DirectShow 应用程序都应该包含 Dshow.h 文件。其实，Dshow.h 中没有什么特别的定义，只是嵌套包含了其他一些头文件：

```
// Include standard Windows files
#include <windows.h>
#include <windowsx.h>
#include <olectl.h>
#include <ddraw.h>
#include <mmsystem.h>
// Include DirectShow include files
#include <strmif.h>  // Generated IDL header file for streams interfaces
#include <amvideo.h>    // ActiveMovie video interfaces and definitions
#include <amaudio.h>    // ActiveMovie audio interfaces and definitions
#include <control.h>    // generated from control.odl
#include <evcode.h>     // event code definitions
#include <uuids.h>    // declaration of type GUIDs and well-known clsids
#include <errors.h>     // HRESULT status and error definitions
#include <edevdefs.h>   // External device control interface defines
#include <audevcod.h>   // audio filter device error event codes
#include <dvdevcod.h>   // DVD error event codes
```

不难发现，上述这些头文件在 streams.h 中也包含了。在实际应用中，我们也常常用 streams.h 文件来替换 Dshow.h 文件。

5.1.2　需要连接的库文件

DirectShow SDK 建议，DirectShow 应用程序应该至少连接库文件 Strmiids.lib 和 Quartz.lib。前者定义了 DirectShow 标准的 CLSID 和 IID，后者定义了导出函数 AMGetErrorText（如果应用程序中没有使用到这个函数，也可以不连接这个库）。如果我们写程序的时候包含了头文件 streams.h，则一般库文件还要连接 strmbasd.lib（Release 版本为 strmbase.lib）、uuid.lib 和 winmm.lib。

提示　*在 VC 中连接库文件的方法是：选择 VC 的菜单项 Project | Settings | Link，在这页中的 Category 选择 General，然后在 Object/library modules 中输入库文件名称。*

5.1.3 VC 的系统编译环境

确认 DirectX SDK 的 Include 目录和 Lib 目录都已经加入了 VC 的系统编译环境，并且放在标准的 VC 目录之前，以保证编译器能够拿到最新版本的源文件。

> 提示
>
> VC 的系统编译环境配置方法，请参见 4.1 节。

因为 DirectShow 应用程序是一种 COM 客户程序，因此在调用任何 COM 函数之前调用 CoInitialize（或 CoInitializeEx）函数进行 COM 库的初始化（一般是在应用程序启动的时候调用一次），在结束 COM 库使用时调用 CoUninitialize 函数进行反初始化（一般是在应用程序退出前调用一次）。

5.2 一般开发过程

开发 DirectShow 应用程序，一般有 3 个阶段，如图 5.1 所示。

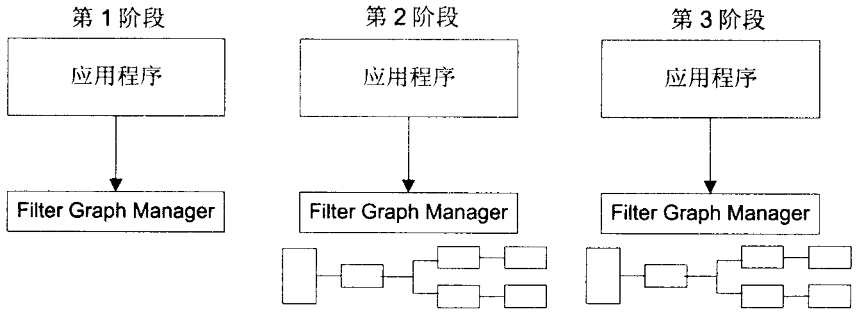

图 5.1 DirectShow 应用程序一般开发过程

第 1 阶段，创建一个 Filter Graph Manager 组件，代码如下：

```
IGraphBuilder *pGraph = NULL;
HRESULT hr = CoCreateInstance(CLSID_FilterGraph, NULL,
CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);
```

第 2 阶段，根据实际的应用，创建一条完整的 Filter 链路。比如播放一个本地文件，最简单快速的代码如下：

```
hr = pGraph->RenderFile(L"C:\\Example.avi", NULL);
```

第 3 阶段，调用 Filter Graph Manager 上（或者直接在某个 Filter 上）的各个接口方法进行控制，并且完成 Filter Graph Manager 与应用程序的事件交互。比如调用 IMediaControl 接口方法控制 Filter Graph 的状态转换，代码如下：

```
IMediaControl *pControl = NULL;
hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
hr = pControl->Run();    // 运行 Filter Graph
```

在第 2 阶段，在我们编程构建 Filter 链路之前，我们怎么来证明构思中的 Filter 链路是切实可行的呢？最常用的方法是，首先在 GraphEdit 中验证，然后再以 GraphEdit 中的 Filter 链路为模型在程序中实现。

在程序中构建完 Filter 链路后，有时候并不会马上能够正常运行。要是能够“直观地”看到在程序中 Filter 是怎么连接的，问题就要好办一点了。实际上，GraphEdit 也提供了这样一种能够显示其他应用程序创建的 Fiter Graph 的功能。

提示

GraphEdit 的这个功能仅在 Windows 2000 或 Windows XP 上支持。

要使用 GraphEdit 的这个功能，在程序中创建了一个 Filter Graph Manager 之后，应该立即将其注册到活动对象表（Running Object Table， ROT）中，代码如下：

```
HRESULT AddToRot(IUnknown *pUnkGraph, DWORD *pdwRegister)
{
    IMoniker * pMoniker;
    IRunningObjectTable *pROT;
    if (FAILED(GetRunningObjectTable(0, &pROT))) {
        return E_FAIL;
    }
    WCHAR wsz[256];
    wsprintfW(wsz, L"FilterGraph %08x pid %08x", (DWORD_PTR)pUnkGraph,
GetCurrentProcessId());
    HRESULT hr = CreateItemMoniker(L"!", wsz, &pMoniker);
    if (SUCCEEDED(hr)) {
        hr = pROT->Register(ROTFLAGS_REGISTRATIONKEEPSALIVE, pUnkGraph,
            pMoniker, pdwRegister);
        pMoniker->Release();
    }
    pROT->Release();
    return hr;
}
```

之后，我们就可以在 GraphEdit 中连接、显示这个 Filter Graph 了。方法为：选择 GraphEdit 的菜单项 File | Connect to remote Graph，在弹出的如图 5.2 所示的对话框中选择一个 Filter Graph 标识项后，单击 OK 按钮，我们就能看到，相应应用程序的 Filter Graph 显示在 GraphEdit 的客户区窗口中了。

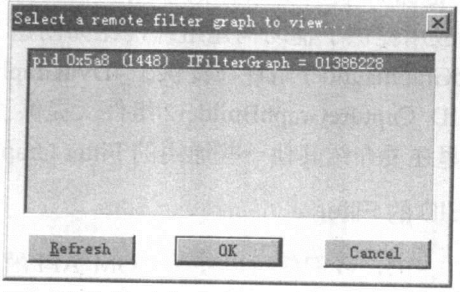

图 5.2 远程连接其他应用程序创建的 Filter Graph

提示 当GraphEdit用于显示其他应用程序的Filter Graph时，请尽量不要使用GraphEdit提供的功能编辑Filter链路（增加或者删除Filter），也不要改变这个Filter Graph的运行状态，否则会引起它的应用程序的不可预知性错误。

还需要提醒大家一点，在我们的应用程序退出前，应该将这个显示了我们应用程序创建的 Filter Graph 的 GraphEdit 程序关掉。在程序中析构 Filter Graph Manager 之前，我们还必须从 ROT 中注销我们先前的注册，代码如下：

```
void RemoveFromRot(DWORD pdwRegister)
{
    IRunningObjectTable *pROT;
    if (SUCCEEDED(GetRunningObjectTable(0, &pROT))) {
        pROT->Revoke(pdwRegister);
        pROT->Release();
    }
}
```

在应用程序中，整个过程实现的代码如下：

```
IGraphBuilder *pGraph;
DWORD dwRegister;
// Create the filter graph manager
CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
                        IID_IGraphBuilder, (void **)&pGraph);
#ifdef _DEBUG
hr = AddToRot(pGraph, &dwRegister);
#endif
// Rest of the application (not shown)
#ifdef _DEBUG
RemoveFromRot(dwRegister);
#endif
pGraph->Release();
```

5.3 通用 Filter Graph 构建技术

构建什么样的 Filter 链路，取决于具体的应用。为了简化 Filter Graph 的构建，DirectShow 为几个典型的应用提供了辅助构建组件，比如创建 DVD 播放的 Filter Graph 时，可以使用 CLSID_DvdGraphBuilder 组件（提供了 IDvdGraphBuilder 接口）；进行音视频采集时，可以使用 CLSID_CaptureGraphBuilder2 组件（提供了 ICaptureGraphBuilder2 接口）。除此之外，我们这里还要介绍其他一些通用的 Filter Graph 构建技术。

5.3.1 加入一个指定 CLSID 的 Filter

给定了一个 Filter 的 CLSID，我们就可以使用 COM API 函数 CoCreateInstance 来创建它，并且使用 IFilterGraph::AddFilter 接口方法将其加入到 Filter Graph 中，代码如下：

```
HRESULT AddFilterByCLSID(
    IGraphBuilder *pGraph,  // Pointer to the Filter Graph Manager
    const GUID& clsid,      // CLSID of the filter to create
    LPCWSTR wszName,        // A name for the filter
    IBaseFilter **ppF)      // Receives a pointer to the filter
{
    if (!pGraph || ! ppF) return E_POINTER;
    *ppF = 0;
    IBaseFilter *pF = 0;
    HRESULT hr = CoCreateInstance(clsid, 0, CLSCTX_INPROC_SERVER,
        IID_IBaseFilter, reinterpret_cast<void**>(&pF));
    if (SUCCEEDED(hr))
    {
        hr = pGraph->AddFilter(pF, wszName);
        if (SUCCEEDED(hr))
            *ppF = pF;
        else
            pF->Release();
    }
    return hr;
}
```

我们现在要向 Filter Graph 中加入一个 AVI Mux Filter，代码如下：

```
IBaseFilter *pMux;
hr = AddFilterByCLSID(pGraph, CLSID_AviDest, L"AVI Mux", &pMux);
if (SUCCEEDED(hr))
{
    /* ... */
   pMux->Release();
}
```

提示　并不是所有的 Filter 都能通过 CoCreateInstance 函数来创建，大部分注册在 Audio Compressors 目录下的 Filter（它们使用了 CLSID_ACMWrapper 包装 Filter）、大部分注册在 Video Compressors 目录下的 Filter（它们使用了 CLSID_AVICo 包装 Filter），以及代表硬件（WDM 或 VFW）的包装 Filter，必须通过枚举的方式来创建。

5.3.2　得到 Filter 上的未连接 Pin

当我们在程序中连接 Filter 的时候，首先是要取得 Filter 上未连接的输入 Pin 或者输出 Pin。方法是，枚举 Filter 上的所有 Pin，调用 IPin::QueryDirection 查看 Pin 的方向，调用 IPin::ConnectedTo 查看 Pin 的连接状态。代码如下：

```
HRESULT GetUnconnectedPin(
    IBaseFilter *pFilter,   // Pointer to the filter
    PIN_DIRECTION PinDir,   // Direction of the pin to find
    IPin **ppPin)           // Receives a pointer to the pin
{
```

```
    *ppPin = 0;
    IEnumPins *pEnum = 0;
    IPin *pPin = 0;
    HRESULT hr = pFilter->EnumPins(&pEnum);
    if (FAILED(hr))
    {
        return hr;
    }
    while (pEnum->Next(1, &pPin, NULL) == S_OK)
    {
        PIN_DIRECTION ThisPinDir;
        pPin->QueryDirection(&ThisPinDir);
        if (ThisPinDir == PinDir)
        {
            IPin *pTmp = 0;
            hr = pPin->ConnectedTo(&pTmp);
            if (SUCCEEDED(hr)) // Already connected, not the pin we want
            {
                pTmp->Release();
            }
            else  // Unconnected, this is the pin we want
            {
                pEnum->Release();
                *ppPin = pPin;
                return S_OK;
            }
        }
        pPin->Release();
    }
    pEnum->Release();
    // Did not find a matching pin
    return E_FAIL;
}
```

比如我们现在要得到一个 Filter 的未连接的输出 Pin，代码如下：

```
IPin *pOut = NULL;
HRESULT hr = GetUnconnectedPin(pFilter, PINDIR_OUTPUT, &pOut);
if (SUCCEEDED(hr))
{
    /* ... */
    pOut->Release();
}
```

5.3.3 连接两个 Filter

两个 Filter 的连接，总是从上一级 Filter 的输出 Pin 指向下一级 Filter 的输入 Pin。Filter Graph Manager 提供的直接连接两个 Filter 的函数为 IFilterGraph::ConnectDirect 和 IGraphBuilder::Connect。参考代码如下：

```
HRESULT ConnectFilters(
    IGraphBuilder *pGraph, // Filter Graph Manager
```

```
    IPin *pOut,              // Output pin on the upstream filter
    IBaseFilter *pDest)      // Downstream filter
{
    if ((pGraph == NULL) || (pOut == NULL) || (pDest == NULL))
    {
        return E_POINTER;
    }
#ifdef debug
        PIN_DIRECTION PinDir;
        pOut->QueryDirection(&PinDir);
        _ASSERTE(PinDir == PINDIR_OUTPUT);
#endif
    // 得到下一级 Filter 的输入 Pin
    IPin *pIn = 0;
    HRESULT hr = GetUnconnectedPin(pDest, PINDIR_INPUT, &pIn);
    if (FAILED(hr))
    {
        return hr;
    }
    // 将输出 Pin 连接到输入 Pin
    hr = pGraph->Connect(pOut, pIn);
    pIn->Release();
    return hr;
}
// 不同参数的 ConnectFilters 函数重载形式
HRESULT ConnectFilters(
    IGraphBuilder *pGraph,
    IBaseFilter *pSrc,
    IBaseFilter *pDest)
{
    if ((pGraph == NULL) || (pSrc == NULL) || (pDest == NULL))
    {
        return E_POINTER;
    }
    // Find an output pin on the first filter
    IPin *pOut = 0;
    HRESULT hr = GetUnconnectedPin(pSrc, PINDIR_OUTPUT, &pOut);
    if (FAILED(hr))
    {
        return hr;
    }
    hr = ConnectFilters(pGraph, pOut, pDest);
    pOut->Release();
    return hr;
}
```

比如我们现在要加入一个 AVI Mux Filter 和一个 File Writer Filter，然后将它们相连起来。代码如下：

```
IBaseFilter *pMux, *pWrite;
hr = AddFilterByCLSID(pGraph, CLSID_AviDest, L"AVI Mux", &pMux);
```

```
    if (SUCCEEDED(hr))
    {
        hr = AddFilterByCLSID(pGraph, CLSID_FileWriter, L"File Writer",
&pWrite);
        if (SUCCEEDED(hr))
        {
            hr = ConnectFilters(pGraph, pMux, pWrite);
            /* Use IFileSinkFilter to set the file name (not shown). */

          pWrite->Release();
        }
        pMux->Release();
    }
```

5.3.4 查找 Filter 或 Pin 上的接口

大部分情况下，我们使用 Filter Graph Manager 上的接口来操作 Filter Graph。但有些时候，我们也需要直接获得 Filter 或者它的 Pin 上的某个接口进行参数配置等。

如果 Filter Graph 中的 Filter 都是我们一个一个加进去的，而且我们知道自己需要的接口具体在哪个 Filter 上实现，我们当然可以直接调用这个 Filter 或它的 Pin 上的 QueryInterface 函数来得到我们想要的接口。

但问题是，如果上述条件不满足，我们该怎么办呢？于是，我们就要枚举 Graph 中的所有 Filter，或者枚举 Filter 上的所有 Pin，一个一个地找。代码如下：

```
// 查找 Filter 上实现的某个接口
HRESULT FindFilterInterface(
    IGraphBuilder *pGraph, // Pointer to the Filter Graph Manager
    REFGUID iid,           // IID of the interface to retrieve
    void **ppUnk)          // Receives the interface pointer
{
    if (!pGraph || !ppUnk) return E_POINTER;
    HRESULT hr = E_FAIL;
    IEnumFilters *pEnum = NULL;
    IBaseFilter *pF = NULL;
    if (FAILED(pGraph->EnumFilters(&pEnum)))
    {
        return E_FAIL;
    }
    // 分别调用 QueryInterface，枚举 Filter Graph 中的所有 Filter
    while (S_OK == pEnum->Next(1, &pF, 0))
    {
        hr = pF->QueryInterface(iid, ppUnk);
        pF->Release();
        if (SUCCEEDED(hr))
        {
            break;
        }
    }
    pEnum->Release();
```

```
    return hr;
}
// 查找给定 Filter 的 Pin 上实现的某个接口
HRESULT FindPinInterface(
    IBaseFilter *pFilter,  // Pointer to the filter to search
    REFGUID iid,           // IID of the interface
    void **ppUnk)          // Receives the interface pointer
{
    if (!pFilter || !ppUnk) return E_POINTER;
    HRESULT hr = E_FAIL;
    IEnumPins *pEnum = 0;
    if (FAILED(pFilter->EnumPins(&pEnum)))
    {
        return E_FAIL;
    }
    // 分别调用 QueryInterface，枚举 Filter 上的所有 Pin
    IPin *pPin = 0;
    while (S_OK == pEnum->Next(1, &pPin, 0))
    {
        hr = pPin->QueryInterface(iid, ppUnk);
        pPin->Release();
        if (SUCCEEDED(hr))
        {
            break;
        }
    }
    pEnum->Release();
    return hr;
}
// 综合了 FindFilterInterface 和 FindPinInterface 两个函数的功能
HRESULT FindInterfaceAnywhere(
    IGraphBuilder *pGraph,
    REFGUID iid,
    void **ppUnk)
{
    if (!pGraph || !ppUnk) return E_POINTER;
    HRESULT hr = E_FAIL;
    IEnumFilters *pEnum = 0;
    if (FAILED(pGraph->EnumFilters(&pEnum)))
    {
        return E_FAIL;
    }
    // Loop through every filter in the graph
    IBaseFilter *pF = 0;
    while (S_OK == pEnum->Next(1, &pF, 0))
    {
        hr = pF->QueryInterface(iid, ppUnk);
        if (FAILED(hr))
        {
            // The filter does not expose the interface, but maybe
```

```
            // one of its pins does
            hr = FindPinInterface(pF, iid, ppUnk);
        }
        pF->Release();
        if (SUCCEEDED(hr))
        {
            break;
        }
    }
    pEnum->Release();
    return hr;
}
```

比如我们现在通过 IGraphBuilder::RenderFile 函数，构建了一条内含 DV 格式数据的 AVI 文件的回放链路。我们想要得到其中 DV 视频解码 Filter 上的 IIPDVDec 接口，以设置 DV 解码输出的图像大小为原始图像的四分之一。实现代码如下：

```
hr = pGraph->RenderFile(L"C:\\Example.avi", 0);
if (SUCCEEDED(hr))
{
    IIPDVDec *pDvDec;
    hr = FindFilterInterface(pGraph, IID_IIPDVDec, (void**)&pDvDec);
    if (SUCCEEDED(hr))
    {
        pDvDec->put_IPDisplay(DVRESOLUTION_QUARTER);
        pDvDec->Release();
    }
    else if (hr == E_NOINTERFACE)
    {
        // This file does not contain DV video
    }
    else
    {
        // Some other error occurred
    }
}
```

5.3.5 遍历 Filter 链路

给定一条 Filter 链路中的某个 Filter，我们可以往上或往下遍历得到所有的其他 Filter。方法是：从本级 Filter 的输入 Pin 或者输出 Pin，可以得到上一级或下一级 Filter 的连接着的 Pin，再调用这个 Pin 的 IPin::QueryPinInfo 就可以得到它的拥有者 Filter。重复上述过程，我们就可以一直遍历到 Source Filter 或者 Renderer Filter。

给定一个 Filter，得到它的往上或往下相连的 Filter 的函数的代码如下：

```
HRESULT GetNextFilter(
    IBaseFilter *pFilter, // 遍历开始的 Filter
    PIN_DIRECTION Dir,     // 遍历方向
    IBaseFilter **ppNext) // 得到的相邻 Filter
{
```

```
    if (!pFilter || !ppNext) return E_POINTER;
    IEnumPins *pEnum = 0;
    IPin *pPin = 0;
    HRESULT hr = pFilter->EnumPins(&pEnum);  // 枚举 Filter 上所有 Pin
    if (FAILED(hr)) return hr;
    while (S_OK == pEnum->Next(1, &pPin, 0))
    {
        // See if this pin matches the specified direction
        PIN_DIRECTION ThisPinDir;
        hr = pPin->QueryDirection(&ThisPinDir);
        if (FAILED(hr))
        {
            // Something strange happened.
            hr = E_UNEXPECTED;
            pPin->Release();
            break;
        }
        if (ThisPinDir == Dir)
        {
            // Check if the pin is connected to another pin
            IPin *pPinNext = 0;
            hr = pPin->ConnectedTo(&pPinNext);
            if (SUCCEEDED(hr))
            {
                // Get the filter that owns that pin
                PIN_INFO PinInfo;
                hr = pPinNext->QueryPinInfo(&PinInfo);
                pPinNext->Release();
                pPin->Release();
                pEnum->Release();
                if (FAILED(hr) || (PinInfo.pFilter == NULL))
                {
                    // Something strange happened.
                    return E_UNEXPECTED;
                }
                // This is the filter we're looking for
                *ppNext = PinInfo.pFilter; // Client must release
                return S_OK;
            }
        }
        pPin->Release();
    }
    pEnum->Release();
    // Did not find a matching filter
    return E_FAIL;
}
```

5.3.6 成批删除 Filter

删除 Filter Graph 中的所有 Filter 的代码如下：

```
// 首先停止 Filter Graph 的运行
pControl->Stop();
// 枚举 Filter Graph 中的所有 Filter
IEnumFilters *pEnum = NULL;
HRESULT hr = pGraph->EnumFilters(&pEnum);
if (SUCCEEDED(hr))
{
    IBaseFilter *pFilter = NULL;
    while (S_OK == pEnum->Next(1, &pFilter, NULL))
     {
        pGraph->RemoveFilter(pFilter);  // 删除 Filter
        pEnum->Reset();                 // 复位枚举器内部状态
        pFilter->Release();
    }
    pEnum->Release();
}
```

递归调用，删除指定 Filter 下游的（连接着的）所有 Filter 的代码如下：

```
void NukeDownstream(IGraphBuilder * inGraph, IBaseFilter * inFilter)
{
  if (inGraph && inFilter)
  {
     IEnumPins * pinEnum = 0;
     if (SUCCEEDED(inFilter->EnumPins(&pinEnum)))
     {
       pinEnum->Reset();
       IPin * pin = 0;
       ULONG cFetched = 0;
       bool pass = true;
       // 枚举 Filter 上的所有 Pin
       while (pass && SUCCEEDED(pinEnum->Next(1, &pin, &cFetched)))
       {
              if (pin && cFetched)
              {
                IPin * connectedPin = 0;
               // 判断当前 Pin 是否处于连接状态
                pin->ConnectedTo(&connectedPin);
                if(connectedPin)
                {
                      PIN_INFO pininfo;
                     if  (SUCCEEDED(connectedPin->QueryPinInfo(&pininfo)))
                      {
                      // 如果当前 Pin 连接着的 Pin 是输入 Pin，则说明我们能够得到
                      // 一个下一级 Filter，我们要递归调用 NukeDownstream
                         if(pininfo.dir == PINDIR_INPUT)
                         {
                         // 这里递归调用
                         NukeDownstream(inGraph, pininfo.pFilter);
                         // 当这一路下游 Filter 全部删除后，可以断开当前 Pin 连接
                         inGraph->Disconnect(connectedPin);
```

```
                        inGraph->Disconnect(pin);
                       // 将操作完成的下一级 Filter 删除
                        inGraph->RemoveFilter(pininfo.pFilter);
                      }
                      pininfo.pFilter->Release();
                    }
                    connectedPin->Release();
                }
                pin->Release();
            }
            else
            {
                    pass = false;
            }
        }
        pinEnum->Release();
    }
  }
}
```

递归调用，删除指定 Filter 上游的（连接着的）所有 Filter 的代码如下：

```
void NukeUpstream(IGraphBuilder * inGraph, IBaseFilter * inFilter)
{
  if (inGraph && inFilter)
  {
    IEnumPins * pinEnum = 0;
    if (SUCCEEDED(inFilter->EnumPins(&pinEnum)))
    {
      pinEnum->Reset();
      IPin * pin = 0;
      ULONG cFetched = 0;
      bool pass = true;
      // 枚举 Filter 上的所有 Pin
      while (pass && SUCCEEDED(pinEnum->Next(1, &pin, &cFetched)))
      {
            if (pin && cFetched)
            {
              IPin * connectedPin = 0;
             // 判断当前 Pin 是否处于连接状态
             pin->ConnectedTo(&connectedPin);
              if (connectedPin)
              {
                    PIN_INFO pininfo;
                    if(SUCCEEDED(connectedPin->QueryPinInfo(&pininfo)))
                    {
                    // 如果当前 Pin 连接着的 Pin 是输出 Pin，则说明我们能够得到
                    // 一个上一级 Filter，我们要递归调用 NukeUpstream
                      if(pininfo.dir == PINDIR_OUTPUT)
                      {
                       // 递归调用
```

```
                    NukeUpstream(inGraph, pininfo.pFilter);
                    // 当这一路上游 Filter 全部删除后，可以断开当前 Pin 连接
                    inGraph->Disconnect(connectedPin);
                    inGraph->Disconnect(pin);
                    // 将操作完成的上一级 Filter 删除
                    inGraph->RemoveFilter(pininfo.pFilter);
                }
                pininfo.pFilter->Release();
            }
            connectedPin->Release();
        }
        pin->Release();
    }
    else
    {
            pass = false;
    }
  }
  pinEnum->Release();
 }
}
}
```

5.4 事件交互的实现

DirectShow 应该有一种机制，能够让应用程序与 Filter Graph Manager 之间实现交互控制。这个机制其实就是事件通知（Event Notification），即当 Filter 状态转换、运行时遇到错误或要求重画视频窗口时，都会发出一个特定的事件，由 Filter Graph Manager 处理或者转发给应用程序，应用程序可以接收事件，并根据事件类型做出相应的处理。

DirectShow 中定义的常见的事件有：EC_COMPLETE，表示 Filter Graph 中所有的数据都已经回放完毕（此时 Filter Graph 不会自动转入停止状态，需要在应用程序接收到这个事件后调用 Filter Graph Manager 的 IMediaControl::Stop）；EC_ERRORABORT，表示 Filter Graph 运行时出错；EC_DEVICE_LOST，表示热插拔设备（典型的如 USB 设备、1394 接口设备等）脱离系统；EC_REPAINT，表示视频窗口要求重画当前图像帧。

提示

更多的事件请参考 SDK 帮助文档的 Event Notification Codes。

下面首先介绍事件处理机制。Filter Graph Manager 上有 3 个接口与事件通知有关，即 IMediaEventSink、IMediaEvent 和 IMediaEventEx。

- IMediaEventSink：用在 Filter 内部，它的接口方法 Notify 用于向 Filter Graph Manager 发送事件通知；
- IMediaEvent：应用程序就是使用它的接口方法来处理 Filter Graph Manager 发出的事件；

➢ IMediaEventEx：IMediaEvent 接口的扩展，支持当 Filter Graph Manager 有事件要发出时，用窗口消息的方式通知应用程序。

事件处理的大致过程如下：

（1）Filter Graph 中的 Filter 发出一个事件（运行时出错了或者满足了一定的条件），接收者为 Filter Graph Manager;

（2）Filter Graph Manager 对一些特殊的事件有默认的处理方法。在接收到事件后，要么按默认方法直接处理这个事件，要么放到一个事件队列中（我们当然也可以通过调用接口方法 IMediaEvent::CancelDefaultHandling 来取消指定事件的默认处理，使 Filter Graph Manager 接收到指定的事件后，直接将其放到事件队列中），等待上层的应用程序处理;

（3）应用程序在获知 Filter Graph Manager 有事件发出后，就可以使用 IMediaEventEx 接口方法从事件队列中读取事件，然后根据事件类型作出相应的处理。

提示　这里所说的事件队列有点类似于 Windows 的消息队列，Filter Graph Manager 与应用程序之间的事件交互过程也跟 Windows 的消息循环差不多。

接下来重点介绍应用程序对事件通知的处理方法。首先要弄明白的是，应用程序怎么知道 Filter Graph Manager 的事件队列中有事件在等待处理？有两种方法，一种是 Filter Graph Manager 通过发送指定的窗口消息通知应用程序，另一种是通过事件同步对象（注意，这里的事件是指 CreateEvent 创建的事件，而不是 Filter 发送的类似于窗口消息的事件）。两种方法相比，第 1 种比较常用，因为 Windows 程序处理窗口消息要容易得多。

第 1 种方法的使用步骤如下:

（1）自定义一个消息，然后调用 IMediaEventEx::SetNotifyWindow 将其设置给 Filter Graph Manager，代码如下：

```
#define WM_GRAPHNOTIFY  (WM_APP + 100)    // Private message.
//...
IMediaEventEx   *pEvent = NULL;
hr = pGraph->QueryInterface(IID_IMediaEventEx, (void **)&pEvent);
hr = pEvent->SetNotifyWindow((OAHWND)m_hwnd, WM_GRAPHNOTIFY, 0);
```

（2）在指定的消息接收窗口类中定义消息映射，代码如下：

```
BEGIN_MESSAGE_MAP(CNotifyWnd, …)
    //{{AFX_MSG_MAP(CNotifyWnd)
    //}}AFX_MSG_MAP
    ON_MESSAGE(WM_ GRAPHNOTIFY, OnGraphNotify)
END_MESSAGE_MAP()
```

（3）在消息响应函数中获取 Filter Graph 的事件通知，并作出相应处理，代码如下：

```
void CNotifyWnd:: OnGraphNotify(WPARAM inWParam, LPARAM inLParam)
{
if (pEvent)
{
    LONG eventCode = 0, eventParam1 = 0, eventParam2 = 0;
```

```
    while (SUCCEEDED(pEvent->GetEvent(&eventCode, &eventParam1, &event-
Param2, 0)))
    {
        pEvent->FreeEventParams(eventCode, eventParam1, eventParam2);
        switch (eventCode)
        {
        case EC_COMPLETE:
            // Do something...
            break;
        case EC_USERABORT:
            // Do something...
            break;
        case EC_ERRORABORT:
            // Do something...
            break;
        default:
            break;
        }
    }
}
}
```

值得注意的是，由于 Filter Graph Manager 使用事件队列，与窗口消息的处理一样都是异步的，所以，在上述步骤（3）中我们看到，当接收到通知消息后使用了 while 语句循环调用 IMediaEvent::GetEvent，直到事件队列中的所有事件处理完毕。另外需要注意的是，当得到事件后，应调用 IMediaEvent::FreeEventParams 释放 Filter Graph Manager 为该事件分配的资源。

提示 *在释放 IMediaEventEx 接口之前，务必再次调用 IMediaEventEx::SetNotifyWindow，将参数设置为 NULL：pEvent>SetNotifyWindow(NULL, 0, 0)。*

第 2 种方法其实也很简单，因为 Filter Graph Manager 在内部创建了一个事件同步对象，它在事件队列中有尚未处理的事件时，状态标记为有效，而当应用程序不断调用接口方法 IMediaEvent::GetEvent 取空事件队列中所有事件之后状态复位。我们可以通过接口方法 IMediaEvent::GetEventHandle 来得到这个事件同步对象的句柄。代码如下：

```
HANDLE  hEvent;
long    evCode, param1, param2;
BOOLEAN bDone = FALSE;
HRESULT hr = S_OK;
hr = pEvent->GetEventHandle((OAEVENT*)&hEvent);
while(!bDone)
{
    if (WAIT_OBJECT_0 == WaitForSingleObject(hEvent, 100))
    {
        while (hr = pEvent->GetEvent(&evCode, &param1, &param2, 0),
SUCCEEDED(hr))
        {
```

```
            printf("Event code: %#04x\n    Params: %d, %d\n", evCode,
param1, param2);
            hr = pEvent->FreeEventParams(evCode, param1, param2);
            bDone = (EC_COMPLETE == evCode);
        }
    }
}
```

需要说明的是，这种方法适用于控制台程序、主程序包含多线程的应用程序，以及其他一些不适合使用窗口消息的场合。

提示　这两种处理事件通知的方法，没有好坏之分，到底选择哪一种方法，完全取决于我们的实际应用环境。

5.5　进度条的实现

当播放媒体文件的时候，怎样才能像微软公司的 Windows 媒体播放器一样，实现播放进度的即时显示、随机位置拖动等？其实，我们可以依赖 Filter Graph Manager 上的 IMediaSeeking 接口。

大致的实现步骤如下（以基于对话框的应用程序为例）：

（1）在 VC 的资源编辑器中为主对话框加入一个如图 5.3 所示的进度控件，并且使用 VC 的类向导（选择菜单项 View | ClassWizard | Member Variables）为这个控件添加一个控制类，代码如下：

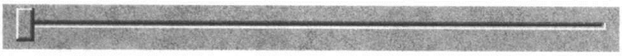

图 5.3　进度控件

```
CSliderCtrl     mSliderGraph;
```

（2）在主对话框的初始化函数 OnInitDialog 中，初始化进度控件，代码如下：

```
mSliderGraph.SetRange(0, 1000);
mSliderGraph.SetPos(0);
```

（3）实现进度的即时显示。用一个定时器，不断地调用 IMediaSeeking::GetCurrentPosition 接口方法获得当前的播放时间点，然后根据媒体文件的总时间长度，计算进度的百分比，再转化为进度控件的当前位置值，调用 CSliderCtrl::SetPos 函数设置。代码如下：

```
// 当 Filter Graph 进入运行状态时，同时启动一个定时器
void CMainDlg::OnButtonPlay()
{
   if (mFilterGraph)
   {
        mFilterGraph->Run();
```

```
            // Start a timer
            if (mSliderTimer == 0)
            {
                mSliderTimer = SetTimer(SLIDER_TIMER, 100, NULL);
            }
        }
}
// 定时器消息处理函数
void CMainDlg::OnTimer(UINT nIDEvent)
{
    if (nIDEvent == mSliderTimer && mFilterGraph)
    {
        double pos = 0, duration = 1.;
        mFilterGraph->GetCurrentPosition(&pos);
        mFilterGraph->GetDuration(&duration);
        // Get the new position, and update the slider
        int newPos = int(pos * 1000 / duration);
        if (mSliderGraph.GetPos() != newPos)
        {
            mSliderGraph.SetPos(newPos);
        }
    }
    CDialog::OnTimer(nIDEvent);
}
```

（4）实现随机位置拖动，这其实是进度的即时显示的反过程。当用户拖动进度块到一个新位置后，我们需要得到这个进度值，然后转化为 Filter Graph 的播放进度时间点，并调用接口方法 IMediaSeeking::SetPositions 进行设置，于是，Filter Graph 就从新的位置继续播放。我们在 WM_HSCROLL 的响应消息中实现上述功能，代码如下：

```
void CMainDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    if (pScrollBar->GetSafeHwnd() == mSliderGraph.GetSafeHwnd())
    {
        if (mFilterGraph)
        {
            // Calculate the current seeking position
            double duration = 1.;
            mFilterGraph->GetDuration(&duration);
            double pos = duration * mSliderGraph.GetPos() / 1000.;
            mFilterGraph->SetCurrentPosition(pos);
        }
    }
    else
    {
        CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
    }
}
```

5.6　Filter 属性页的显示

在大多数商业软件中，并不会将 Filter 的属性页直接显示给用户。因为属性页的界面过于简单（有时候还可能是很随意的），它们主要是用作 Filter 接口的调试的。

不过在这里，我们还是要介绍一种使用 OleCreatePropertyFrame 函数，在应用程序中直接显示 Filter 的属性页的方法，代码如下：

```
IBaseFilter *pFilter;
/* Obtain the filter's IBaseFilter interface. (Not shown) */
ISpecifyPropertyPages *pProp = NULL;
HRESULT  hr  =  pFilter->QueryInterface(IID_ISpecifyPropertyPages,  (void
**)&pProp);
if (SUCCEEDED(hr))
{
    // Get the filter's name and IUnknown pointer.
    FILTER_INFO FilterInfo;
    hr = pFilter->QueryFilterInfo(&FilterInfo);
    IUnknown *pFilterUnk;
    pFilter->QueryInterface(IID_IUnknown, (void **)&pFilterUnk);
    // Show the page
    CAUUID caGUID;
    pProp->GetPages(&caGUID);
    pProp->Release();
    OleCreatePropertyFrame(
        hWnd,                   // Parent window
        0, 0,                   // Reserved
        FilterInfo.achName,     // Caption for the dialog box
        1,                      // Number of objects (just the filter)
        &pFilterUnk,            // Array of object pointers.
        caGUID.cElems,          // Number of property pages
        caGUID.pElems,          // Array of property page CLSIDs
        0,                      // Locale identifier
        0, NULL                 // Reserved
    );
    // Clean up
    pFilterUnk->Release();
    FilterInfo.pGraph->Release();
    CoTaskMemFree(caGUID.pElems);
}
```

5.7　系统设备的枚举

我们知道，DirectShow 对系统中的不可预知性硬件提供了良好的支持。既然是不可预知的，我们在应用程序中到底怎么来支持呢？答案是使用系统枚举。

系统枚举的一般过程如图 5.4 所示。

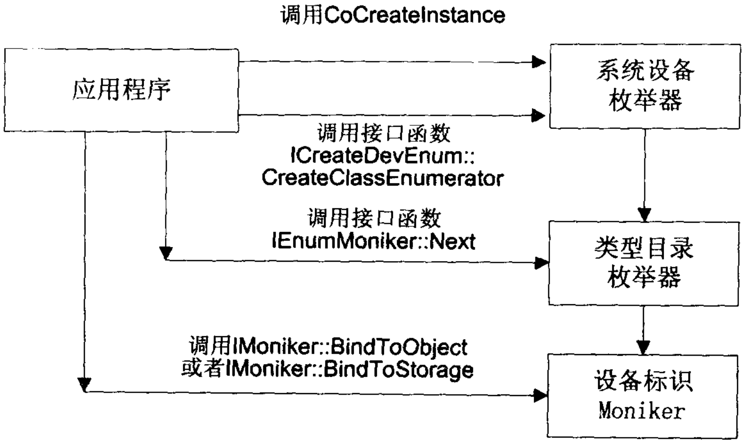

图 5.4 系统枚举过程

（1）使用 CoCreateInstance 函数创建系统枚举器组件对象（CLSID 为 CLSID_System-DeviceEnum），并获得 ICreateDevEnum 接口；

（2）使用接口方法 ICreateDevEnum::CreateClassEnumerator 为指定的 Filter 注册类型目录创建一个枚举器，并获得 IEnumMoniker 接口；

（3）使用 IEnumMoniker 接口方法枚举指定类型目录下所有设备标识（Device Moniker）；

（4）调用 IMoniker::BindToStorage 之后，可以访问设备标识的属性集，比如得到 Display Name、Friendly Name 等；

（5）调用 IMoniker::BindToObject 可以将设备标识生成一个 DirectShow Filter，随后调用 IFilterGraph::AddFilter，并将之加入到 Filter Graph 中就可以参与工作了。

下面是实现这个过程的代码：

```
// 创建一个系统枚举组件
HRESULT hr;
ICreateDevEnum *pSysDevEnum = NULL;
hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_INPROC_SERVER,
    IID_ICreateDevEnum, (void **)&pSysDevEnum);
if (FAILED(hr))
{
    return hr;
}
// 指定枚举的类型目录，获得 IEnumMoniker 接口
IEnumMoniker *pEnumCat = NULL;
hr = pSysDevEnum->CreateClassEnumerator(CLSID_VideoInputDeviceCategory,
&pEnumCat, 0);
if (hr == S_OK)
{
    // 使用 IEnumMoniker 接口枚举所有的设备标识
    IMoniker *pMoniker = NULL;
    ULONG cFetched;
```

```
    while(pEnumCat->Next(1, &pMoniker, &cFetched) == S_OK)
    {
        IPropertyBag *pPropBag;
        hr = pMoniker->BindToStorage(0, 0, IID_IPropertyBag,
            (void **)&pPropBag);
        if (SUCCEEDED(hr))
        {
            // 获得 Filter 的 Friendly Name
            VARIANT varName;
            VariantInit(&varName);
            hr = pPropBag->Read(L"FriendlyName", &varName, 0);
            if (SUCCEEDED(hr))
            {
                // Display the name in your UI somehow
            }
            VariantClear(&varName);
            // 创建 Filter 实例
            IBaseFilter *pFilter;
            hr = pMoniker->BindToObject(NULL, NULL, IID_IBaseFilter,
                (void**)&pFilter);
            // Now add the filter to the graph
            // Remember to release pFilter later
            pPropBag->Release();
        }
        pMoniker->Release();
    }
    pEnumCat->Release();
}
pSysDevEnum->Release();
```

提示　从设备标识中一般可以获得两种名字：Display Name 和 Friendly Name。前者可以通过 IMoniker::GetDisplayName 接口函数直接得到，是一串带有 CLSID 的复杂字符，一般不直接显示给用户；而后者是一种容易让用户从字面理解的字符串，一般在界面元素中显示。

知道了一个硬件的设备标识，我们也可以直接通过 IFilterGraph2::AddSourceFilterForMoniker 接口方法来创建 Filter，代码如下：

```
LPOLESTR strName = NULL;
IBaseFilter pSrc = NULL;
hr = pMoniker->GetDisplayName(NULL, NULL, &strName);
if (SUCCEEDED(hr))
{
    // 获得 IFilterGraph2 接口
    IFilterGraph2 *pFG2 = NULL;
    hr = pGraph->QueryInterface(IID_IFilterGraph2, (void**)&pFG2);
    if (SUCCEEDED(hr))
    {
        // 根据设备标识创建一个 Source Filter
```

```
            hr = pFG2->AddSourceFilterForMoniker(pMoniker, 0, L"Source",
&pSrc);
            pFG2->Release();
        }
        CoTaskMemFree(strName);
    }
    // If successful, remember to release pSrc
```

或者给定一个 Filter 类型目录的 Display Name 描述，格式为@device:*:{category-clsid}，调用 IMoniker::ParseDisplayName 或 MkParseDisplayName 函数对字符串进行分析，创建一个该类型目录下第一个返回的设备标识对象。代码如下：

```
// 视频采集设备的类型目录
WCHAR szMon[] = L"@device:*:{860BB310-5D01-11d0-BD3B-00A0C911CE86}";
IBindCtx *pBindCtx;
hr = CreateBindCtx(0, &pBindCtx);
ULONG chEaten = 0;
IMoniker *pMoniker = 0;
hr = MkParseDisplayName(pBindCtx, szMon, &chEaten, &pMoniker);
pBindCtx->Release();
if (SUCCEEDED(hr))
{
    // Get the display name, or bind to a DirectShow filter
    // ...
    pMoniker->Release();
}
```

5.8 图片的抓取

在播放媒体文件的过程中，有一个很有用的功能，就是在当前播放的位置抓取图片。实现这种图片抓取功能的方法很多，我们这里只介绍常用的两种。

第 1 种方法最简单，它使用 IBasicVideo::GetCurrentImage 接口方法，代码如下。

```
bool SnapshotBitmap(IBasicVideo * pBasicVideo, const char * outFile)
{
   if (pBasicVideo)
   {
          long bitmapSize = 0;
          // 首先获得图像大小
          if (SUCCEEDED(pBasicVideo->GetCurrentImage(&bitmapSize, 0)))
          {
                 bool pass = false;
                 // 分配图像帧内存
                 unsigned char * buffer = new unsigned char[bitmapSize];
                 // 获取图像帧数据
                 if  (SUCCEEDED(pBasicVideo->GetCurrentImage(&bitmapSize,
(long *)buffer)))
                 {
```

```
                    BITMAPFILEHEADER    hdr;
                    LPBITMAPINFOHEADER  lpbi;
                    lpbi = (LPBITMAPINFOHEADER)buffer;
                    int nColors = 1 << lpbi->biBitCount;
                    if (nColors > 256)
                        nColors = 0;
                    // always is "BM"
                    hdr.bfType          = ((WORD) ('M' << 8) | 'B');
                    hdr.bfSize          = bitmapSize + sizeof( hdr );
                    hdr.bfReserved1     = 0;
                    hdr.bfReserved2     = 0;
                    hdr.bfOffBits       = (DWORD)
    (sizeof(BITMAPFILEHEADER) + lpbi->biSize + nColors * sizeof(RGBQUAD));
                    CFile bitmapFile(outFile, CFile::modeReadWrite |
CFile::modeCreate | CFile::typeBinary);
                    // 写入位图文件头
                    bitmapFile.Write(&hdr, sizeof(BITMAPFILEHEADER));
                    // 写入图像帧数据（包括 BITMAPINFOHEADER 信息）
                    bitmapFile.Write(buffer, bitmapSize);
                    bitmapFile.Close();
                    pass = true;
                }
                delete [] buffer;
                return pass;
            }
        }
        return false;
    }
```

值得注意的是，IBasicVideo 接口应该从 Filter Graph Manager 上获得，但真正实现在 Renderer Filter 上。如果我们使用的是传统的 Video Renderer，那么使用 GetCurrentImage 抓图将是不可靠的。因为如果 Video Renderer 使用了 DirectDraw 加速，这个函数调用会失败；而且调用这个函数，Video Renderer 必须处于暂停状态。但如果我们使用的是 VMR，则没有上述这些限制。

第 2 种方法比较复杂，它使用 Sample Grabber Filter。它其实是一个 Trans-In-Place Filter，在 SDK 安装目录下的 Samples\C++\DirectShow\Filters\Grabber 提供了源代码。实际上，Sample Grabber 可以抓取任何类型的 Sample。但在这里，我们只介绍使用它抓取视频帧的方法。步骤如下：

（1）创建 Sample Grabber，并将之加入到 Filter Graph 中。

```
// Create the Sample Grabber
IBaseFilter *pGrabberF = NULL;
hr = CoCreateInstance(CLSID_SampleGrabber, NULL, CLSCTX_INPROC_SERVER,
    IID_IBaseFilter, (void**)&pGrabberF);
if (FAILED(hr))
{
    // Return an error
}
```

```
hr = pGraph->AddFilter(pGrabberF, L"Sample Grabber");
if (FAILED(hr)
{
    // Return an error
}
ISampleGrabber *pGrabber = NULL;
pGrabberF->QueryInterface(IID_ISampleGrabber, (void**)&pGrabber);
```

（2）给 Sample Grabber 设置 Pin 上连接用的媒体类型。

如果我们想抓取 24 位的 RGB 图片，如下设置媒体类型：

```
AM_MEDIA_TYPE mt;
ZeroMemory(&mt, sizeof(AM_MEDIA_TYPE));
mt.majortype = MEDIATYPE_Video;
mt.subtype = MEDIASUBTYPE_RGB24;
hr = pGrabber->SetMediaType(&mt);
```

也可以根据当前显示器的配置来设置 Sample Grabber 接受的 RGB 类型，代码如下：

```
// Find the current bit depth
HDC hdc = GetDC(NULL);
int iBitDepth = GetDeviceCaps(hdc, BITSPIXEL);
ReleaseDC(NULL, hdc);
// Set the media type
mt.majortype = MEDIATYPE_Video;
switch (iBitDepth)
{
case 8:
    mt.subtype = MEDIASUBTYPE_RGB8;
    break;
case 16:
    mt.subtype = MEDIASUBTYPE_RGB555;
    break;
case 24:
    mt.subtype = MEDIASUBTYPE_RGB24;
    break;
case 32:
    mt.subtype = MEDIASUBTYPE_RGB32;
    break;
default:
    return E_FAIL;
}
hr = pGrabber->SetMediaType(&mt);
```

（3）完成 Filter Graph 的构建。

因为 Sample Grabber 上已经设置了一个媒体类型，则其他 Filter 必须以这种媒体类型才能与 Sample Grabber 相连。我们可以使用 DirectShow 的“智能连接”机制，来简化这个 Fitler Graph 的创建过程，代码如下。

```
IBaseFilter *pSrc;
hr = pGraph->AddSourceFilter(wszFileName, L"Source", &pSrc);
```

```
if (FAILED(hr))
{
    // Return an error code
}
hr = ConnectFilters(pGraph, pSrc, pGrabberF);
```

其中，ConnectFilters 是我们在 5.3 节中介绍的自定义函数。

如果我们只是想抓图（不需要对视频预览），则 Sample Grabber 后面可以连接一个 Null Renderer Filter（它的 CLSID 为 CLSID_NullRenderer）。如果要 Filter Graph 中的数据流以最快的速度传送，则 Filter Graph 不要使用参考时钟（调用 IMediaFilter::SetSyncSource，参数为 NULL）。

提示　我们不推荐 Sample Grabber 后面连接标准的 Video Renderer。因为 Sample Grabber 是一个 Trans-In-Place Filter，输入和输出 Pin 使用相同的数据缓存，而且这个内存很可能位于显卡上。而从显存中读取数据比直接从主机内存中读取数据要慢得多，这将大大降低视频图像显示的性能。

（4）运行 Filter Graph。

Sample Grabber 可以有如下两种工作模式：

缓冲模式　将输入的 Sample 进行缓存后，再往下传送。

回调模式　当有输入的 Sample 时，调用应用程序设置进来的回调函数。

因为回调模式会影响整个 Filter Graph 的效率，并且容易引起死锁，所以我们推荐使用缓冲模式。另外，我们可以设置 ISampleGrabber::SetOneShot，使得 Sample Grabber 获取一个 Sample 以后，就让 Filter Graph 停止，代码如下：

```
// Set one-shot mode and buffering.
hr = pGrabber->SetOneShot(TRUE);
hr = pGrabber->SetBufferSamples(TRUE);
pControl->Run(); // Run the graph.
pEvent->WaitForCompletion(INFINITE, &evCode); // Wait till it's done.
```

（5）获取抓到的 Sample 数据。缓冲模式下，我们可以调用 ISampleGrabber::GetCurrentBuffer 来获取 Sample 数据，代码如下：

```
// Find the required buffer size
long cbBuffer = 0;
hr = pGrabber->GetCurrentBuffer(&cbBuffer, NULL);
char *pBuffer = new char[cbBuffer];
if (!pBuffer)
{
    // Out of memory. Return an error code
}
hr = pGrabber->GetCurrentBuffer(&cbBuffer, (long*)pBuffer);
```

我们也可以将获取的数据使用 GDI 函数显示出来，代码如下：

```
AM_MEDIA_TYPE mt;
hr = pGrabber->GetConnectedMediaType(&mt);
```

```
if (FAILED(hr))
{
    // Return error code
}
// Examine the format block
VIDEOINFOHEADER *pVih;
if ((mt.formattype == FORMAT_VideoInfo) &&
    (mt.cbFormat >= sizeof(VIDEOINFOHEADER)) &&
    (mt.pbFormat != NULL) )
{
    pVih = (VIDEOINFOHEADER*)mt.pbFormat;
}
else
{
    // Wrong format. Free the format block and return an error.
    FreeMediaType(mt);
    return VFW_E_INVALIDMEDIATYPE;
}
// You can use the media type to access the BITMAPINFOHEADRE information.
// For example, the following code draws the bitmap using GDI:
SetDIBitsToDevice(
    hdc, 0, 0,
    pVih->bmiHeader.biWidth,
    pVih->bmiHeader.biHeight,
    0, 0,
    0,
    pVih->bmiHeader.biHeight,
    pBuffer,
    (BITMAPINFO*)&pVih->bmiHeader,
    DIB_RGB_COLORS
);
// Free the format block when you are done:
FreeMediaType(mt);
```

5.9 一个简单的媒体文件播放器

通过以上的学习，我们现在着手编写一个简单的媒体文件播放器。这个播放器的界面如图 5.5 所示。

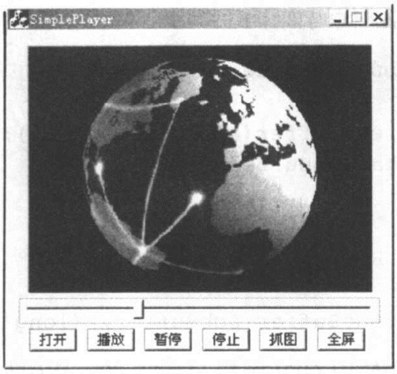

图 5.5 简单媒体文件播放器界面

通过“打开”按钮，可以调入多种媒体文件；“播放”、“暂停”、“停止”各按钮分别控制媒体文件的播放、暂停和停止；“抓图”用于抓取播放过程中的当前图像帧，并保存为一个本地 BMP 文件；“全屏”用于全屏播放文件（在全屏模式下按下 Esc 或 Enter 键可以恢复到正常模式）等。

下面，我们简单介绍一下这个播放器的开发过程。

（1）使用 VC 的项目向导生成一个基于对话框的 MFC 应用程序，项目名为 SimplePlayer；

（2）配置 VC 的连接库，即增加 strmbasd.lib（Release 版本为 strmbase.lib）、uuid.lib 和 winmm.lib，并如下添加 COM 库的初始化、反初始化代码：

```
BOOL CSimplePlayerApp::InitInstance()
{
    CoInitialize(NULL);     // 初始化 COM 库的使用
    CSimplePlayerDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
           // TODO: Place code here to handle when the dialog is
           //  dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
           // TODO: Place code here to handle when the dialog is
           //  dismissed with Cancel
    }
    // Since the dialog has been closed, return FALSE so that we exit the
    //  application, rather than start the application's message pump
    return FALSE;
}
int CSimplePlayerApp::ExitInstance()
{
    CoUninitialize();

    return CWinApp::ExitInstance();
}
```

（3）在资源编辑器中编辑主对话框界面，见图 5.5；

（4）定义一个视频窗口的控制对象，代码如下：

```
CStatic    mVideoWindow;
```

为了让视频窗口正常刷新，我们还必须修改视频窗口的风格，即在主对话框的初始化函数 OnInitDialog 中增加如下函数调用：

```
mVideoWindow.ModifyStyle(0, WS_CLIPCHILDREN);
```

还要添加主对话框的 WM_ERASEBKGND 消息响应，代码如下：

```
BOOL CSimplePlayerDlg::OnEraseBkgnd(CDC* pDC)
{
    // Intercept background erasing for the movie window, since the
    // video renderer will keep the screen painted.  Without this code,
    // your video window might get painted over with gray (the default
    // background brush) when it is obscured by another window and
    // redrawn
    CRect rc;
    // Get the bounding rectangle for the movie screen
    mVideoWindow.GetWindowRect(&rc);
    ScreenToClient(&rc);
    // Exclude the clipping region occupied by our movie screen
    pDC->ExcludeClipRect(&rc);
    // Erase the remainder of the dialog as usual
    return CDialog::OnEraseBkgnd(pDC);
}
```

（5）把 Filter Graph 的各种相关操作封装成一个类 CDXGraph，包括 Filter Graph Manager 的创建、其上面的各种控制接口等。在主对话框类中，定义一个 CDXGraph 类型的指针，代码如下：

```
CDXGraph *   mFilterGraph;
```

（6）根据用户选择的源文件，创建相应的回放 Filter Graph，代码如下：

```
// 析构当前使用的 Filter Graph
void CSimplePlayerDlg::DestroyGraph(void)
{
    if (mFilterGraph)
    {
            // Stop the filter graph first
            mFilterGraph->Stop();
            mFilterGraph->SetNotifyWindow(NULL);

            delete mFilterGraph;
            mFilterGraph = NULL;
    }
}
// 创建新的 Filter Graph
void CSimplePlayerDlg::CreateGraph(void)
{
    DestroyGraph();
    mFilterGraph = new CDXGraph();
    if (mFilterGraph->Create())
    {
            // Render the source clip
            mFilterGraph->RenderFile(mSourceFile);
            // Set video window and notification window
            mFilterGraph->SetDisplayWindow(mVideoWindow.GetSafeHwnd());
            mFilterGraph->SetNotifyWindow(this->GetSafeHwnd());
            // Show the first frame
            mFilterGraph->Pause();
```

```
    }
}
```

（7）处理 Filter Graph 的事件。

定义一个消息映射，代码如下：

```
BEGIN_MESSAGE_MAP(CSimplePlayerDlg, CDialog)
    //{{AFX_MSG_MAP(CSimplePlayerDlg)
    // ...
    //}}AFX_MSG_MAP
    ON_MESSAGE(WM_GRAPHNOTIFY, OnGraphNotify)
END_MESSAGE_MAP()
```

声明一个消息响应函数，代码如下：

```
afx_msg LRESULT OnGraphNotify(WPARAM inWParam, LPARAM inLParam);
```

实现代码如下：

```
LRESULT CSimplePlayerDlg::OnGraphNotify(WPARAM inWParam, LPARAM inLParam)
{
    IMediaEventEx * pEvent = NULL;
    if (mFilterGraph && (pEvent = mFilterGraph->GetEventHandle()))
    {
            LONG   eventCode = 0, eventParam1 = 0, eventParam2 = 0;
            while (SUCCEEDED(pEvent->GetEvent(&eventCode, &eventParam1,
&eventParam2, 0)))
            {
                    // Spin through the events
                    pEvent->FreeEventParams(eventCode, eventParam1,
eventParam2);

                    switch (eventCode)
                    {
                    case EC_COMPLETE:
                            OnButtonPause();
                            mFilterGraph->SetCurrentPosition(0);
                            break;
                    case EC_USERABORT:
                    case EC_ERRORABORT:
                            OnButtonStop();
                            break;
                    default:
                            break;
                    }
            }
    }
    return 0;
}
```

（8）实现进度条、随机拖动，可以参考 5.5 节。

提示

下载地址（见前言）中的 SimplePlayer 项目，提供了这个媒体文件播放器的完整实现。

第6章 音频采集

6.1 应用分析

现在的所谓多媒体电脑一般都会有声卡（软声卡或硬声卡），有声卡就能进行音频采集。大家一定都用过 Windows 系统自带的附件“录音机”，如图 6.1 所示，它可以通过麦克风进行录音，最终生成一个 Wave 文件。

图 6.1 Windows 自带录音机程序界面

其实，“录音”就是属于音频采集的范畴。只不过 Windows 的“录音机”可能是直接使用一系列 API 函数来实现的，而我们现在想要在 DirectShow 中实现。

6.1.1 应用方案

大家知道，DirectShow 对硬件的支持是通过特定的包装 Filter 来实现的。这里，声卡使用的是 Audio Capture Filter，它内部又使用了以 waveIn 开头的一套 API 函数来实现（如 waveInOpen 等）。

运行 GraphEdit 并选择插入 Filter 时，在 Audio Capture Sources 目录下，我们就能看到所有代表本地机器上的声卡的各个 Filter（如果机器装了多张声卡，这里就会有多个 Filter），如图 6.2 所示。在 Filter Graph 中加入这个 Filter，我们会发现这个 Filter 有很多输入 Pin，如 Line In、CD Audio、Microphone、Stereo Mix 等，还有一个叫 Capture 的输出 Pin。

提示

在 Filter Graph 中，Audio Capture Filter 的所有输入 Pin 上并没有真正的数据流入，而仅仅是声卡的各个输入端子的象征性表示。我们在对某个输入端子进行音频采集时，应该确认这个输入端子的物理连接，而不要在 Filter Graph 中去连接这个输入 Pin。

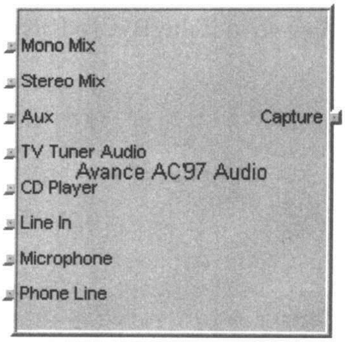

图 6.2　Audio Capture Filter

那么在 DirectShow 中，我们怎么来开发音频采集应用程序呢？首先，当然是要创建一个 Audio Capture Filter。但大家知道，不同的机器可能有不同的声卡，我们的应用程序不应该指定了去创建某个品牌的声卡 Filter。于是，我们使用了 DirectShow 的系统设备枚举。因为代表声卡的 Filter 都被注册为 CLSID_AudioInputDeviceCategory 类型。使用系统设备枚举器枚举这个类型，就能找到我们想要创建的声卡对象。（枚举时，我们第 1 个得到的将是系统“控制面板”中的“声音和多媒体”设置的首选录音音频设备。）

提示

如何进行枚举并创建代表声卡的 Filter，我们已经在 5.7 节中做过详细的介绍，这里就不再赘述。

当成功加入声卡 Filter 后，接下去的问题就是要构建一个完整的 Filter Graph 了。这因应用而异。比如，我们现在想采集生成一个 Wave 文件，那么我们还需加入一个 Wave Dest Filter 和一个 File Writer Filter，然后依次将它们相连。

提示

Wave Dest Filter 是 DirectShow SDK 中的一个 Filter 例子，位于 SDK 安装目录的 Samples\C++\DirectShow\Filters\WavDest 下，主要完成的功能是，在结束采集时，即 PCM 数据全部写入文件之后，再在文件头位置插入一个 Wave 格式的数据块。我们必须首先编译这个例子项目并且注册这个 Filter 后，才能使用到本应用中。

在 GraphEdit 中的 Filter 连接，如图 6.3 所示。

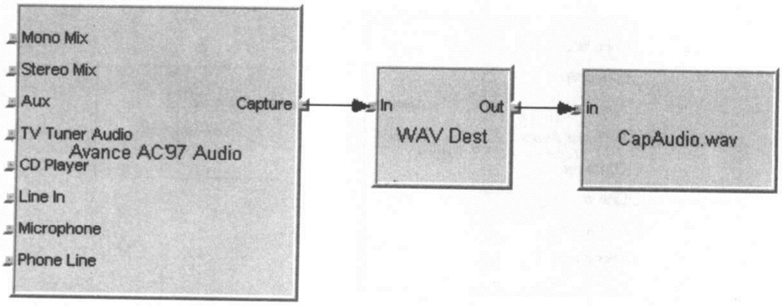

图 6.3　音频采集到 Wave 文件

创建音频采集程序的框架代码（AddFilterByClsid 和 ConnectTwoFilters 都是我们在 5.3 节中介绍的自定义函数）如下：

```
void BuildAudioCaptureGraph(void)  // Warning! No error checking here
{
    IBaseFilter *pSrc = NULL, *pWaveDest = NULL, *pWriter = NULL;
    IFileSinkFilter *pSink= NULL;
    IGraphBuilder *pGraph;
    // Create the Filter Graph Manager
    CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
        IID_IGraphBuilder, (void**)&pGraph);
    // Add the audio capture filter
    FindAudioCapture(&pSrc); // Assume that this function enumerates
                             // audio capture devices and picks one
    pGraph->AddFilter(pSrc, L"Capture");
    // Add the WavDest and the File Writer
    AddFilterByClsid(pGraph, L"WavDest", CLSID_WavDest, &pWavDest);
    AddFilterByClsid(pGraph, L"File Writer", CLSID_FileWriter, &pWriter);
    // Set the file name
    pWriter->QueryInterface(IID_IFileSinkFilter, (void**)&pSink);
    pSink->SetFileName(L"C:\\CapAudio.wav", NULL);
    // Hook everything up
    ConnectTwoFilters(pGraph, pSrc, pWavDest);
    ConnectTwoFilters(pGraph, pWavDest, pWriter);
}
```

音频采集除了生成 Wave 文件外，我们还可以做其他的应用，比如生成一个 AVI 文件或 ASF 文件等。这两种应用的 Filter Graph 如图 6.4 和 6.5 所示。

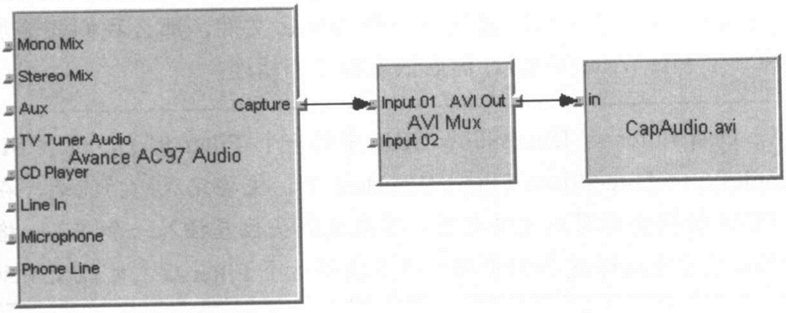

图 6.4　音频采集到 AVI 文件

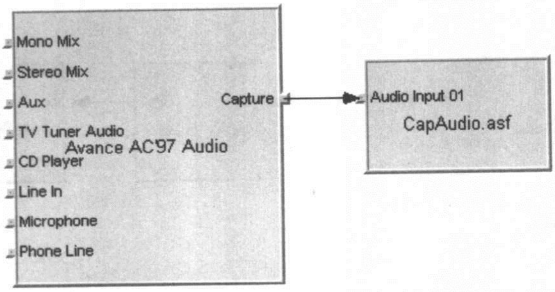

图 6.5　音频采集到 ASF 文件

提示

图 6.5 中用于写文件的 Filter 是 WM ASF Writer（CLSID 为 CLSID_WMAsfWriter），而图 6.3 和图 6.4 中使用的是 File Writer（CLSID 为 CLSID_FileWriter）。

当然，在进行音频采集的同时，我们还可以实时监听音频源的输入，如图 6.6 所示，我们在 Audio Capture Filter 后面接了一个 Infinite Pin Tee，这个 Filter 能够将一个输入 Pin 上输入的数据复制成多份，然后分别通过各个输出 Pin 发送出去。在 Tee Filter 后面，一路连到了 AVI Mux 用于存储文件，另一路连到了 DirectSound Renderer，完成声音到声卡上的播放。

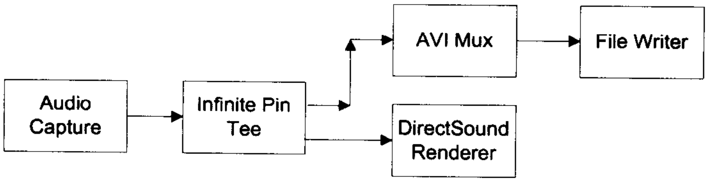

图 6.6　音频采集时实时监听

提示

Infinite Pin Tee 也是 DirectShow SDK 中的一个 Filter 例子，位于 SDK 安装目录的 Samples\C++\DirectShow\Filters\InfTee 下。我们必须首先编译这个例子项目并且注册这个 Filter 后，才能使用到本应用中。

6.1.2　开发要点

在音频采集过程中，我们还会碰到一些接口参数的设置问题。比如，在声卡 Filter 的每个输入 Pin 上，我们都可以得到 IAMAudioInputMixer 接口。通过这个接口，我们可以设置各个输入端子的音频属性，包括进行音频合成时是否允许某个输入端子的音频参与混合、音频输入的音量、高音、低音参数等。另外，在 Filter 上也可以得到 IAMAudioInputMixer 接口，这时调用接口方法就可以统一控制各个输入端子的属性。

音频采集之前，我们还可以通过 Capture 输出 Pin 上的 IAMStreamConfig 接口，设置音频的采样频率以及声音的具体格式（量化精度 8 位或 16 位，单声道或双声道）。代码如下：

```
HRESULT hr = pCapturePin->QueryInterface(IID_IAMStreamConfig, (void
            **)&pCfg);
// Read current media type/format
AM_MEDIA_TYPE *pmt={0};
hr = pCfg->GetFormat(&pmt);
if (SUCCEEDED(hr))
{
        // Fill in values for the new format
        WAVEFORMATEX *pWF = (WAVEFORMATEX *) pmt->pbFormat;
        pWF->nChannels        = (WORD) nChannels;
        pWF->nSamplesPerSec   = nFrequency;
        pWF->nAvgBytesPerSec  = lBytesPerSecond;
```

```
        pWF->wBitsPerSample   = (WORD) (nBytesPerSample * 8);
        pWF->nBlockAlign      = (WORD) (nBytesPerSample * nChannels);
        // Set the new formattype for the output pin
        hr = pCfg->SetFormat(pmt);
        DeleteMediaType(pmt);
    }
    // Release interfaces
    pCfg->Release();
```

另外一点，也是音频采集应用中比较特殊的地方：我们可以通过 Capture 输出 Pin 上的 IAMBufferNegotiation 接口，改变音频采集缓冲的大小，以减少声音的延迟。默认情况下，Audio Capture Filter 使用 0.5s 的缓冲。但对于一些特殊的应用，这么大的缓冲是没有必要的，它带来的延迟也是不能容忍的。一般缓冲设置成能够容纳 80ms 的数据已经很可靠了，甚至 30~40ms 也已经足够了。但是也不能太小，否则会影响音频采集的效率，严重时会损害音质。下面的代码用于设置音频采集的缓冲大小，可供参考：

```
pCapturePin->QueryInterface(IID_IAMBufferNegotiation, (void **)&pNeg);
// Set the buffer size based on selected settings
ALLOCATOR_PROPERTIES prop={0};
prop.cbBuffer = lBufferSize;
prop.cBuffers = 6;
prop.cbAlign = nBytesPerSample * nChannels;
hr = pNeg->SuggestAllocatorProperties(&prop);
pNeg->Release();
```

6.2 实例解剖

DirectShow SDK 提供了一个音频采集的例子，这个例子位于 SDK 安装目录的 Samples\C++\DirectShow\Capture\AudioCap 下。

6.2.1 实现的功能

如图 6.7 所示是 AudioCap 的程序界面。这个应用程序完成的主要功能有：

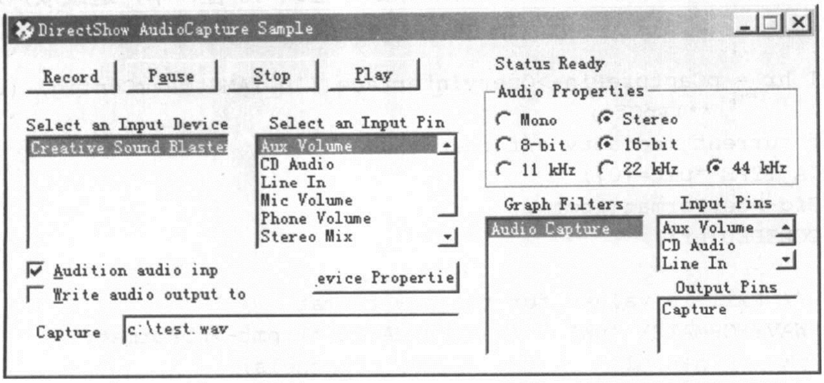

图 6.7 AudioCap 的程序界面

- 枚举系统中的音频采集设备，生成列表；
- 支持采集音频格式的声道、量化精度、采样频率等参数设置；
- 选择一种音频设备后创建录音应用的 Filter Graph（最终生成 Wave 文件），并支持输入端子选择、边听边录等功能；
- 对采集后生成的声音文件进行播放；
- 枚举 Filter Graph 中的所有 Filter，枚举 Filter 上的所有 Pin；
- 显示 Audio Capture Filter 的属性页。

6.2.2 实现要点

通过对 AudioCap 的学习，我们将掌握如下知识点：

- 构建“采集”应用的 Filter Graph；
- 枚举音频采集设备并生成列表；
- 枚举 Filter Graph 中的所有 Filter，并显示 Filter 上的各个 Pin；
- 实现边听边录；
- 使用“智能连接”构建 Filter Graph；
- 使用 IAMAudioInputMixer 接口选择音频采集输入端子；
- 使用 IAMStreamConfig、IAMBufferNegotiation 接口设置采集音频参数；
- 判断 Filter 是否有属性页，以及如何显示属性页。

构建“采集”应用的 Filter Graph

这里所说的“采集”应用，包括音频采集和视频采集。因为有时候采集应用的 Filter Graph 是比较复杂的。DirectShow 专门提供了一个辅助组件 Capture Graph Builder（它的 CLSID 为 CLSID_CaptureGraphBuilder2），来简化这种 Filter Graph 的构建。

这个辅助组件的使用也很简单。首先还是创建一个 Filter Graph Manager，然后创建 Capture Graph Builder 组件并获得 ICaptureGraphBuilder2 接口，接着就是调用接口方法 ICaptureGraphBuilder2::SetFiltergraph 设置 Filter Graph Manager 对象指针。在下面的应用中，我们就可以使用 ICaptureGraphBuilder2 的接口方法（比如 RenderStream、FindPin、FindInterface 等）方便地构建 Filter Graph 了。AudioCap 的实现代码如下：

```
HRESULT CAudioCapDlg::GetInterfaces()
{
    HRESULT hr;
    // Create the filter graph
    hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC,
                          IID_IGraphBuilder, (void **)&m_pGB);
    if (FAILED(hr) || !m_pGB)
        return E_NOINTERFACE;
    // Create the capture graph builder
    hr = CoCreateInstance(CLSID_CaptureGraphBuilder2, NULL,
CLSCTX_INPROC, IID_ICaptureGraphBuilder2, (void **)&m_pCapture);
    if (FAILED(hr) || !m_pCapture)
        return E_NOINTERFACE;
```

```
    // Associate the filter graph with the capture graph builder
    hr = m_pCapture->SetFiltergraph(m_pGB);
    if (FAILED(hr))
        return hr;
    // Get useful interfaces from the graph builder
    JIF(hr = m_pGB->QueryInterface(IID_IMediaControl, (void **)&m_pMC));
    JIF(hr = m_pGB->QueryInterface(IID_IMediaEventEx, (void **)&m_pME));

    // Have the graph signal events via window callbacks
    hr = m_pME->SetNotifyWindow((OAHWND)m_hWnd, WM_GRAPHNOTIFY,
RECORD_EVENT);
CLEANUP:
    return hr;
}
```

枚举音频采集设备并生成列表

AudioCap例子使用了一个列表控件，来显示通过枚举得到的所有音频采集设备。在程序启动的时候，即调用CAudioCapDlg::FillLists函数填写这个列表，内部还嵌套调用了一系列在mfcutil.cpp中实现的工具函数，如EnumFiltersWithMonikerToList、EnumFiltersAndMonikersToList、AddFilterToListWithMoniker等。通过对这些函数的学习，我们将可以掌握整个系统设备的枚举过程。

枚举 Filter Graph 中的所有 Filter，并显示 Filter 上的各个 Pin

AudioCap 例子在 CAudioCapDlg::UpdateFilterLists 函数中填写 Filter Graph 中的所有 Filter 列表，并同步更新指定 Filter 的所有 Pin 列表。在实现中使用了一系列 mfcutil.cpp 的工具函数，如 AddGraphFiltersToList、AddFilterPinsToLists 等。

实现边听边录

我们知道，一般 Audio Capture Filter 只有一个 Capture 输出 Pin，那么怎么来实现边听边录呢（那样至少需要两路音频流）？前面介绍的一种方法是，手动插入一个 Infinite Pin Tee Filter。其实，使用 ICaptureGraphBuilder2::RenderStream 实现起来更简单。它会根据需要自动插入一个 Smart Tee Filter（一个类似于 Infinite Pin Tee 的 DirectShow 标准 Filter）。下面是 AudioCap 例子的 CAudioCapDlg::RenderPreviewStream 函数实现：

```
HRESULT CAudioCapDlg::RenderPreviewStream()
{
    USES_CONVERSION;
    HRESULT hr=0;
    WCHAR wszFilter[128];
    // If we've already configured the stream, just exit
    if (m_pRenderer)
        return S_OK;

    // Set the requested audio properties - buffer sizes, channels, freq, etc
    hr = SetAudioProperties();
    // Render the preview stream
```

```
    hr = m_pCapture->RenderStream(
            &PIN_CATEGORY_PREVIEW,
            &MEDIATYPE_Audio,
            m_pInputDevice,
            NULL,   // No compression filter.
            NULL    // Default renderer.
        );
    // Some capture sources will have only a Capture pin (no Preview pin)
    // In that case, the Capture Graph builder will automatically insert
    // a SmartTee filter, which will split the stream into a Capture
    // stream and a Preview stream.  In that case, it's not an error
    if (hr == VFW_S_NOPREVIEWPIN)
    {
        wcscpy(wszFilter, T2W(TEXT("Smart Tee\0")));
        // Get the interface for the Splitter filter for later use
        hr = m_pGB->FindFilterByName(wszFilter, &m_pSplitter);
        if (FAILED(hr))
            return hr;
    }
    else if (FAILED(hr))
        return hr;
    // Get the interface for the audio renderer filter for later use
    wcscpy(wszFilter, T2W(TEXT("Audio Renderer\0")));
    hr = m_pGB->FindFilterByName(wszFilter, &m_pRenderer);

    return hr;
}
```

使用“智能连接”构建 Filter Graph

DirectShow 的“智能连接”是很聪明的，它能自动使用 Fitler Graph 中未连接的 Filter 进行连接，必要时还会插入新的 Filter。AudioCap 例子在构建采集 Filter 链路时，就使用了这种技术。CAudioCapDlg::RenderCaptureStream 函数的实现代码如下：

```
HRESULT CAudioCapDlg::RenderCaptureStream()
{
    USES_CONVERSION;
    HRESULT hr=0;
    IFileSinkFilter2 *pFileSink;
    // Create the WAVDest filter
    hr = CoCreateInstance(CLSID_WavDest, NULL, CLSCTX_INPROC,
                          IID_IBaseFilter, (void **)&m_pWAVDest);
    if (FAILED(hr))
    {
        return E_NOINTERFACE;
    }
    // Create the FileWriter filter
    hr = CoCreateInstance(CLSID_FileWriter, NULL, CLSCTX_INPROC,
                          IID_IFileSinkFilter2, (void **)&pFileSink);
    if (FAILED(hr))
        return E_NOINTERFACE;
```

```
    // Get the file sink interface from the File Writer
    hr = pFileSink->QueryInterface(IID_IBaseFilter, (void
**)&m_pFileWriter);
    if (FAILED(hr))
        return E_NOINTERFACE;
    // Add the WAVDest filter to the graph
    hr = m_pGB->AddFilter(m_pWAVDest, L"WAV Dest");
    if (FAILED(hr))
        return hr;
    // Add the FileWriter filter to the graph
    hr = m_pGB->AddFilter((IBaseFilter *)m_pFileWriter, L"File Writer");
    if (FAILED(hr))
        return hr;
    // Set filter to always overwrite the file
    hr = pFileSink->SetMode(AM_FILE_OVERWRITE);
    // Set the output filename, which must be a wide string
    TCHAR szEditFile[MAX_PATH];
    WCHAR wszFilename[MAX_PATH];
    m_StrFilename.GetWindowText(szEditFile, MAX_PATH);
    if (szEditFile[0] == TEXT('\0'))
        wcsncpy(wszFilename, T2W(DEFAULT_FILENAME),
NUMELMS(wszFilename)-1);
    else
        wcsncpy(wszFilename, T2W(szEditFile), NUMELMS(wszFilename)-1);
    wszFilename[MAX_PATH-1] = 0;     // Ensure NULL termination
    hr = pFileSink->SetFileName(wszFilename, NULL);
    if (FAILED(hr))
        return hr;
    // Get the pin interface for the capture pin
    IPin *pPin=0;
    if (m_pSplitter)
        hr = GetPin(m_pSplitter, PINDIR_OUTPUT, 0, &pPin);
    else
        hr = GetPin(m_pInputDevice, PINDIR_OUTPUT, 0, &pPin);
    // Connect the new filters
    if (pPin)
    {
        hr = m_pGB->Render(pPin);
        pPin->Release();
    }
    // Release the FileSinkFilter2 interface, since it's no longer needed
    SAFE_RELEASE(pFileSink);
    return hr;
}
```

我们首先创建 Wave Dest Filter 和 File Writer Filter，并将之加入到 Filter Graph 中，经过必要的设置后，取得声卡 Filter 或 Smart Tee 的输出 Pin，即可调用 IGraphBuilder::Render 进行剩余部分的 Filter 连接。非常方便！

使用 IAMAudioInputMixer 接口选择音频采集输入端子

使用接口方法 IAMAudioInputMixer::put_Enable 可以选择输入端子，即枚举所有输入 Pin，分别在各个 Pin 上获取 IAMAudioInputMixer 接口，选中的输入端子调用 put_Enable(TRUE)，其它的都调用 put_Enable(FALSE)。详细实现请参考 AudioCap 例子的 CAudioCapDlg::ActivateSelectedInputPin 函数。

使用 IAMStreamConfig、IAMBufferNegotiation 接口设置采集音频参数

从声卡 Filter 的 Capture 输出 Pin 上可以获取这两个接口。IAMStreamConfig 实际上是给输出 Pin 设置一个采集时输出数据的媒体类型，具体格式数据块用 WAVEFORMATEX 数据结构来描述，描述的内容包括音频的声道数、量化精度、采样频率等。IAMBufferNegotiation 设置的缓冲大小应该根据具体的音频格式来计算，一般设置成能容纳 50ms 数据的缓冲大小已经足够了。详细实现请参考 AudioCap 例子的 CAudioCapDlg::SetAudioProperties 函数。

判断 Filter 是否有属性页以及属性页的显示

我们可以通过 dshowutil.cpp 中实现的工具函数 SupportsPropertyPage，来判断给定的 Filter 上是否有属性页，也就是查看 Filter 上是否支持 ISpecifyPropertyPages 接口，代码如下：

```
BOOL SupportsPropertyPage(IBaseFilter *pFilter)
{
    HRESULT hr;
    ISpecifyPropertyPages *pSpecify;
    // Discover if this filter contains a property page
    hr = pFilter->QueryInterface(IID_ISpecifyPropertyPages, (void
**)&pSpecify);
    if (SUCCEEDED(hr))
    {
        pSpecify->Release();
        return TRUE;
    }
    else
        return FALSE;
}
```

显示属性页同样可以参考 dshowutil.cpp 中的工具函数 ShowFilterPropertyPage 的实现。

第 7 章　视频采集

7.1　应用分析

视频采集的范畴很广，应用也很多，我们这里要介绍的是一种从一些硬件设备中获取视频数据的应用。这些硬件设备包括视频采集卡、电视接收卡、摄像头、录像机（VTR）等。

大家已经知道，DirectShow 对硬件的支持是通过特定的包装 Filter 来实现的。当运行 GraphEdit 并选择插入 Filter 时，在 Video Capture Sources 目录下，我们就能看到所有代表本地机器上的视频采集卡的各个 Filter。在 Filter Graph 中加入其中一个 Filter，我们会发现这个 Filter 一般有一个输入 Pin，还有好几个输出 Pin，如图 7.1 所示。

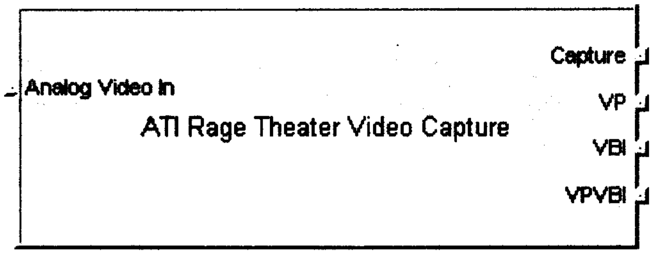

图 7.1　ATI 采集卡的 Capture Filter

一般一个用于数据采集的 Filter 至少有一个称为 Capture 的输出 Pin。典型的情况下，还有一个 Preview Pin 或者 Video Port Pin（VP Pin，支持在内核模式下视频数据到显存的直接传输。这个 Pin 实现预览的效率非常高，一般不会与 Preview Pin 共存，而且即使在不需要预览功能的情况下也必须连接）。有些采集卡能够同时采集视频和音频，那么它的 Filter 自然还应该有音频部分的输出 Pin。

提示

Capture Pin 和 Preview Pin 同样都是输出采集的视频数据的，但用途不同。前者主要用于写文件，后者主要用于显示。另外，它们之间的区别还有：

（1）为了保证 Capture Pin 的输出性能，Preview Pin 会自动丢帧；

（2）从 Capture Pin 出来的 Sample 都打上了采集时刻的时间戳，而从 Preview Pin 出来的 Sample 不打时间戳（原因见 2.9 节）。

DirectShow 支持多种接口的视频采集卡，它们可能以 PCI 或 AGP 的方式直接插入 PC 机箱，也可能以 USB（或 USB2）接口的方式外挂，还有一些设备通过 1394 接口与 PC 机相连（典型的如数码摄像机）。为了保持向后兼容性，DirectShow 同时支持 VFW 驱动的采集卡。

7.1.1　WDM 与 VFW

说起视频采集卡，我们先来看看它的分类。根据使用驱动程序类型的不同，目前市场上大致有两种采集卡：VFW（Video for Windows）卡和 WDM（Windows Driver Model）卡。前者是一种趋于废弃的驱动模型，而后者是前者的替代模型。另外，WDM 还支持更多新的特性，比如直接支持电视接收、视频会议、1394 接口的设备、桌面摄像机、多条视频流（Line-21 或 Closed-Caption 等）同时输出等。（采用 VFW 驱动程序的一般都是一些以前生产的采集卡；市面上新出现的采集卡大都采用了 WDM 驱动程序。）

从编程实现的角度看，对 WDM 卡选择视频输入端子，设置采集输出的图像格式，设置图像的对比度、亮度、色调、饱和度等显示参数，都是通过 COM 接口 IAMCrossbar、IAMStreamConfig 和 IAMVideoProcAmp 来实现的。但对于 VFW 卡，我们却不能编程实现上述设置，而必须将驱动程序定制的设置对话框直接显示给用户（让用户在这些对话框上做出选择）。VFW 驱动程序定制的设置对话框包括 Video Source（设置图像源属性）、Video Format（设置图像输出格式）和 Video Display（设置图像显示参数）。

DirectShow 使用 WDM Video Capture Filter 支持 WDM 卡，而使用另外一个 VFW Capture Filter（它的 CLSID 为 CLSID_VfwCapture）来支持 VFW 卡。代表视频采集卡的 Filter 都被注册为 CLSID_VideoInputDeviceCategory 类型，它们都要靠系统设备枚举来创建（具体过程见 5.7 节）。

有时候，系统中可能装了多张采集卡。我们需要将各个采集卡的名字显示在界面上，让用户选择一个实际使用的设备。因此，在设备的枚举过程中，我们还应得到设备标识对象的一些属性（调用设备标识对象上的接口方法 IMoniker::BindToStorage 后，我们能够得到一个 IPropertyBag 接口，通过 IPropertyBag::Read 调用，就可以读取各个属性了）。常用的属性见表 7.1。

表 7.1　标识对象的一些属性

属性名	含义
FriendlyName	设备的友善名字（用户能够理解）
Description	设备的补充描述（用户能够理解）
DevicePath	设备惟一的字符串（用户很难理解）

提示

FriendlyName 是每个设备都支持的属性；Description 仅仅对 DV 或 DVHS/MPEG 摄像机有效，而且必须有 Windows ME 或 Windows XP 操作系统的支持，它描述的内容比 FriendlyName 更具体，比如可以得到摄像机的生产厂商信息等；DevicePath 用于惟一地标识设备实例，一般只在程序中使用，用户很难理解这种字符串的含义。

读取设备属性的代码如下：

```
HWND hList; // Handle to the list box
IEnumMoniker *pEnum;  // An enumerator for the video capture category
//...
```

```
IMoniker *pMoniker = NULL;
while (pEnum->Next(1, &pMoniker, NULL) == S_OK)
{
    IPropertyBag *pPropBag;
    hr = pMoniker->BindToStorage(0, 0, IID_IPropertyBag,
        (void**)(&pPropBag));
    if (FAILED(hr))
    {
        pMoniker->Release();
        continue;  // Skip this one, maybe the next one will work
    }
    // Find the description or friendly name
    VARIANT varName;
    VariantInit(&varName);
    hr = pPropBag->Read(L"Description", &varName, 0);
    if (FAILED(hr))
    {
        hr = pPropBag->Read(L"FriendlyName", &varName, 0);
    }
    if (SUCCEEDED(hr))
    {
        // Add it to the application's list box
        USES_CONVERSION;
        (long)SendMessage(hList, LB_ADDSTRING, 0,
            (LPARAM)OLE2T(varName.bstrVal));
        VariantClear(&varName);
    }
    pPropBag->Release();
    pMoniker->Release();
}
```

VFW Capture Filter 上支持一个 IAMVfwCaptureDialogs 接口。在 DirectShow 中，我们可以通过这个接口方法来显示 VFW 卡的设置对话框，步骤如下：

（1）停止当前的 Filter Graph；

（2）从采集卡 Filter 上获得 IAMVfwCaptureDialogs 接口；

（3）调用接口方法 IAMVfwCaptureDialogs::HasDialog，询问采集卡驱动程序是否支持指定类型的设置对话框。返回 S_OK 表示支持，返回 S_FALSE 表示不支持；参数取值为枚举类型 VfwCaptureDialog_Source、VfwCaptureDialog_Format 或 wCaptureDialog_Display，分别代表上述 3 种 VFW 设置对话框；

（4）调用 VfwCaptureDialogs::ShowDialog 将指定的对话框显示出来；

（5）重新启动 Filter Graph。

整个过程的代码如下：

```
pControl->Stop(); // Stop the graph
// Query the capture filter for the IAMVfwCaptureDialogs interface
IAMVfwCaptureDialogs *pVfw = 0;
hr = pCap->QueryInterface(IID_IAMVfwCaptureDialogs, (void**)&pVfw);
```

```
if (SUCCEEDED(hr))
{
    // Check if the device supports this dialog box
    if (S_OK == pVfw->HasDialog(VfwCaptureDialog_Source))
    {
        // Show the dialog box
        hr = pVfw->ShowDialog(VfwCaptureDialog_Source, hwndParent);
    }
}
pControl->Run();
```

在程序中如何判断当前枚举得到的是 WDM 卡还是 VFW 卡？

因为 WDM 卡和 VFW 卡使用了不同的包装 Filter，这两种 Filter 上实现的接口自然也有所不同。我们可以试着查询两种卡各自独特的接口，来判断卡的类型。代码如下：

```
BOOL CDsDemoDlg::IsVFWCard(IBaseFilter * pDeviceFilter)
{
    ASSERT(pDeviceFilter);
    IAMVfwCaptureDialogs * pVfw = NULL;
    HRESULT hr = pDeviceFilter->QueryInterface(IID_IAMVfwCaptureDialogs,
(void**)&pVfw);
    if (SUCCEEDED(hr))
    {
        // This is a VFW card...
        pVfw->Release();

        return TRUE;
    }
    else
    {
        // Maybe this is a WDM card...

        return FALSE;
    }
}
BOOL CDsDemoDlg::IsWDMCard(IBaseFilter * pDeviceFilter)
{
    ASSERT(pDeviceFilter);
    IAMAnalogVideoDecoder * pWdm = NULL;
    HRESULT hr = pDeviceFilter->QueryInterface(IID_IAMAnalogVideoDecoder,
(void**)&pWdm);
    if (SUCCEEDED(hr))
    {
        // This is a WDM card...
        pWdm->Release();

        return TRUE;
    }
    else
```

```
    {
        // Maybe this is a VFW card...
        return FALSE;
    }
}
```

7.1.2 构建 Filter Graph

采集应用的 Filter Graph 一般比较复杂；而直接使用 Filter Graph Manager 上的 IGraphBuilder 接口构建这种 Filter Graph，有时候难度又很大。为此，DirectShow 特别提供了一个辅助组件 Capture Graph Builder，来简化这种 Filter Graph 的创建。我们曾经在第 6 章中提到过这个组件的使用，在这里强调一下它的应用。

跟其他 DirectShow 应用程序一样，在构建实际的 Filter Graph 之前，肯定是要创建 Filter Graph Manager 组件的。另外，我们还可以创建一个 Capture Graph Builder 组件，利用这个组件上的 ICaptureGraphBuilder2 接口来完成 Filter Graph 的构建，如图 7.2 所示。

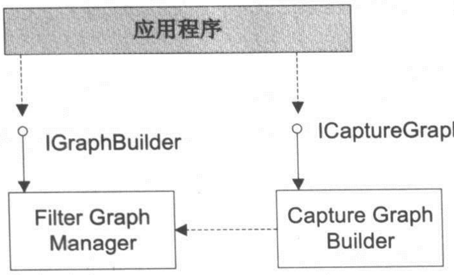

图 7.2　创建 Capture Graph Builder 组件

成功创建 Capture Graph Builder 组件之后，我们还必须对这个组件进行初始化，即调用接口方法 ICaptureGraphBuilder2::SetFilterGraph(pGraph)将 Filter Graph Manager 对象指针设置给它，如图 7.3 所示。

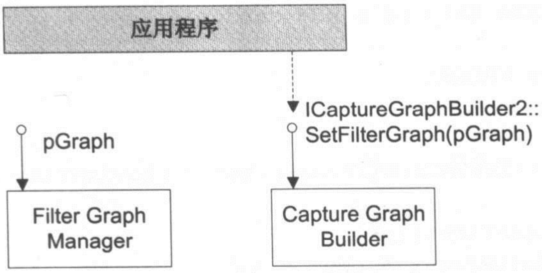

图 7.3　Capture Graph Builder 组件的初始化

Capture Graph Builder 组件进行初始化的代码如下：

```
HRESULT InitCaptureGraphBuilder(
  IGraphBuilder **ppGraph,          // Receives the pointer
  ICaptureGraphBuilder2 **ppBuild   // Receives the pointer
)
{
    if (!ppGraph || !ppBuild)
```

```
    {
        return E_POINTER;
    }
    IGraphBuilder *pGraph = NULL;
    ICaptureGraphBuilder2 *pBuild = NULL;
    // Create the Capture Graph Builder
    HRESULT hr = CoCreateInstance(CLSID_CaptureGraphBuilder2, NULL,
        CLSCTX_INPROC_SERVER, IID_ICaptureGraphBuilder2, (void**)
&pGraph);
    if (SUCCEEDED(hr))
    {
        // Create the Filter Graph Manager
        hr = CoCreateInstance(CLSID_FilterGraph, 0, CLSCTX_INPROC_SERVER,
            IID_IGraphBuilder, (void**)&pGraph);
        if (SUCCEEDED(hr))
        {
            // Initialize the Capture Graph Builder
            pBuild->SetFiltergraph(pGraph);
            // Return both interface pointers to the caller
            *ppBuild = pBuild;
            *ppGraph = pGraph; // The caller must release both
interfaces
            return S_OK;
        }
        else
        {
            pBuild->Release();
        }
    }
    return hr; // Failed
}
```

使用 Capture Graph Builder 组件辅助构建 Filter Graph 的好处有：

- 使用 ICaptureGraphBuilder2::RenderStream 能够对指定类型的 Pin 完成后续连接（比如 PIN_CATEGORY_PREVIEW 指定 Preview Pin，PIN_CATEGORY_CAPTURE 指定 Capture Pin 等）；
- 智能构建 Preview Pin 链路（指定 PIN_CATEGORY_PREVIEW 类型调用 RenderStream），即当采集卡 Filter 含有 VP Pin 时，连接 Overlay Mixer2 | Video Renderer；含有 Preview Pin 时，直接连接 Video Renderer；而只有 Capture Pin 时，自动插入一个 Smart Tee Filter 进行分路后，再连接到 Video Renderer；
- 自动加入采集卡 Filter 之前的辅助 Filter，如用于电视接收的 TV Tuner Filter、用于控制电视声音的 TV Audio Filter 和用于选择视频输入端子的 Analog Video Crossbar Filter 等，并完成这些 Filter 之间的连接（这些 Fitler 之间的连接不是通过媒体类型来标识的，而是通过一种称为媒介（Medium）的 GUID 来标识的，它们之间在 Filter Graph 中也不会有实际的数据流动）；
- ICaptureGraphBuilder2::FindPin 提供了方便的寻找 Pin 的方法；

- ICaptureGraphBuilder2::FindInterface 提供了方便的寻找特定接口的方法；
- 使用 ICaptureGraphBuilder2::ControlStream 可以单独地控制某个 Pin（Preview Pin 或 Capture Pin）；
- 自动处理 EC_REPAINT 问题（如果在采集写文件时，我们同时使用传统的 Video Renderer 进行预览，当视频窗口被别的窗口覆盖然后恢复时，Video Renderer 会向 Filter Graph Manager 发送一个 EC_REPAINT 事件要求重画视频窗口。Filter Graph Manager 的默认处理是，对 Filter Graph 执行暂停操作，这将引起写到文件的内容异常。Capture Graph Builder 会自动帮我们处理这个问题，即它内部调用了 IMediaEvent::CancelDefaultHandling(EC_REPAINT)）。

比如，我们现在想用 ICaptureGraphBuilder2 接口构建预览的 Filter 链路，代码如下：

```
ICaptureGraphBuilder2 *pBuild; // Capture Graph Builder
// Initialize pBuild (not shown)
IBaseFilter *pCap; // Video capture filter
/* Initialize pCap and add it to the filter graph (not shown). */
hr = pBuild->RenderStream(&PIN_CATEGORY_PREVIEW, &MEDIATYPE_Video,
    pCap, NULL, NULL);
```

比如，想要同时构建一个采集写文件的 Filter 链路，代码如下：

```
IBaseFilter *pMux;
hr = pBuild->SetOutputFileName(
    &MEDIASUBTYPE_Avi,   // Specifies AVI for the target file
    L"C:\\Example.avi", // File name
    &pMux,               // Receives a pointer to the mux
    NULL);              // (Optional) Receives a pointer to the file sink
hr = pBuild->RenderStream(
    &PIN_CATEGORY_CAPTURE, // Pin category
    &MEDIATYPE_Video,      // Media type
    pCap,                  // Capture filter
    NULL,                  // Intermediate filter (optional)
    pMux);                 // Mux or file sink filter
// Release the mux filter
pMux->Release();
```

最终构建的 Fitler Graph 如图 7.4 所示。

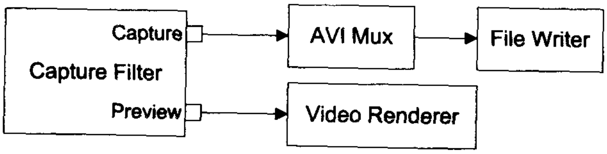

图 7.4　采集写文件的同时预览的 Filter Graph

7.1.3　模拟电视接收

有些采集卡单纯是为了视频采集用的，而有些则还带有电视接收功能。我们这里说的电视，仅限于模拟电视，通过天线接收或有线电缆输入。

DirectShow 对电视接收提供了直接的支持。但电视接收与一般的视频采集有些不同：

- 需要硬件支持频道选择、模拟电视信号接收，并且随后进行采集、数字化；
- 电视声音加载在模拟信号上，需要送到声卡采集、播放（有些电视卡导出一根外部连线输入声卡的 Line In，另外一些通过内部连线作为 CD Input 输入）；
- 某些频道的电视信号可能还带有 VBI（Vertical Blanking Interval）附加信息，如 CC（Closed Captions）、WST（World Standard Teletext）和 XDS（eXtended Data Services）等。

带有电视接收功能的 Filter Graph 如图 7.5 所示。

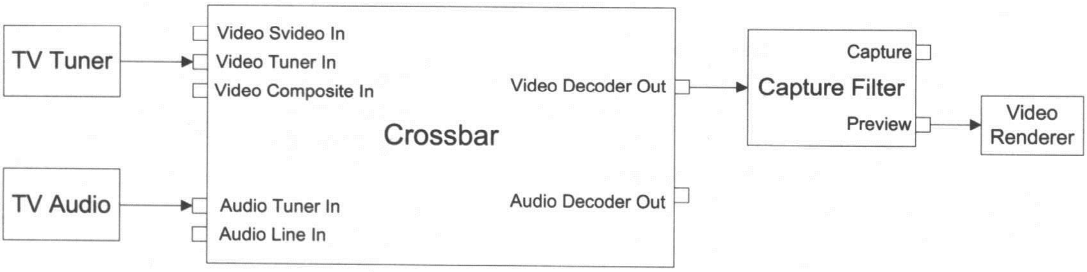

图 7.5　支持电视接收的 Filter Graph

其中，TV Tuner 用于电视频道控制与接收，TV Audio 用于电视声音的解码输出，Crossbar 用于采集卡输入端子的选择（常见的输入端子有 3 种：Video Tuner、Video Composite 和 S-Video），Capture Filter 即采集卡 Filter。

在实际的 Filter Graph 中，我们可以打开 TV Tuner Filter 的属性页进行频道的搜索和设定，如图 7.6 所示。

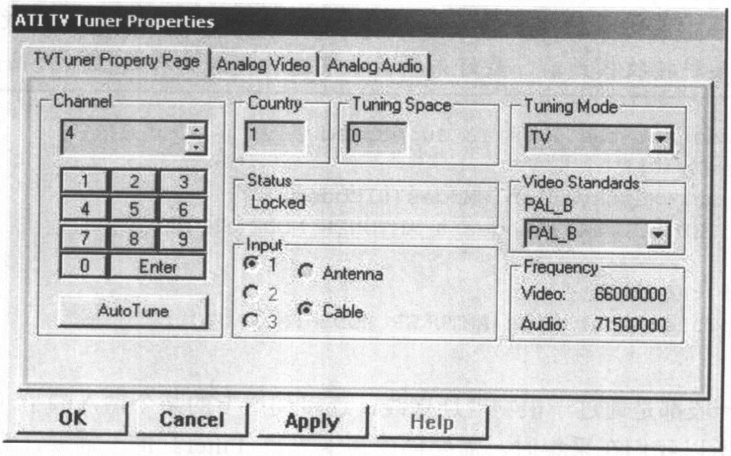

图 7.6　TV Tuner Filter 的属性页

简单的设定方法是：在 Country 文本框输入电视所属的国家代码（中国为 86）；根据实际接到采集卡电视输入端子的信号来源，选择 Antenna（天线）或 Cable（有线电视）；单击 AutoTune 按钮，可以进行全频道搜索。搜索完成后，可以在 Channel 文本框

输入频道序号进行选台。

> **提示** 电视接收后，为了在 Capture Filter 进行正确的解码，我们需要根据实际电视信号的制式（中国为 PAL），设置 Capture Filter 的解码制式。

那么，上面的这些操作，怎么在程序中实现呢（一般的商业软件不推荐将 TV Tuner Filter 的属性页直接显示给用户）？

答案是使用 TV Tuner Filter 上的 IAMTVTuner 接口（这个接口是从 IAMTuner 继承而来的）。我们首先要获得这个接口，代码如下：

```
IAMTVTuner *pTuner = NULL;
hr = pBuild->FindInterface(
    &LOOK_UPSTREAM_ONLY,  // Look upstream from pCap
    NULL,                 // No particular media type
    pCap,                 // Pointer to the capture filter
    IID_IAMTVTuner, (void**)&pTuner);
if (SUCCEEDED(hr))
{
    // Use pTuner ...
    pTuner->Release();
}
```

然后使用 IAMTVTuner 的各个接口方法，如通过 IAMTuner::put_CountryCode 设置国家代码，通过 IAMTVTuner::AutoTune 进行全频道搜索（搜索完成后调用 IAMTVTuner::Store-AutoTune 保存频道-频率映射信息），通过 IAMTVTuner::get_TVFormat 得到当前接收的电视信号制式（需要设置给 Capture Filter），通过 IAMTuner::put_Channel 进行选台等。

> **提示** 有些采集卡支持多种信号的接收，比如电视信号、无线电调频、无线电调幅等。在设置某种接收模式前，最好先检验一下这种模式的有效性，代码如下：

```
// Check whether the mode is supported
long lModes = 0;
hr = m_pTuner->GetAvailableModes(&lModes);
if (SUCCEEDED(hr) && (lModes & AMTUNER_MODE_FM_RADIO))
{
    // Set the mode
    hr = pTuner->put_Mode(AMTUNER_MODE_FM_RADIO);
}
```

电视声音一般都是通过一根物理连接线，连接到声卡的某个输入端子（Line In 或 CD Input）上的，所以我们在采集时，需要创建声卡采集 Filter。惟一需要注意的是，我们没有必要在 Filter Graph 中将 Crossbar 的 Audio Decoder Out Pin 连接到声卡 Filter 的输入 Pin 上，而只需要在代表实际输入端子的 Pin 上调用 IAMAudioInputMixer::put_Enable(TRUE) 就行了，如图 7.7 所示。

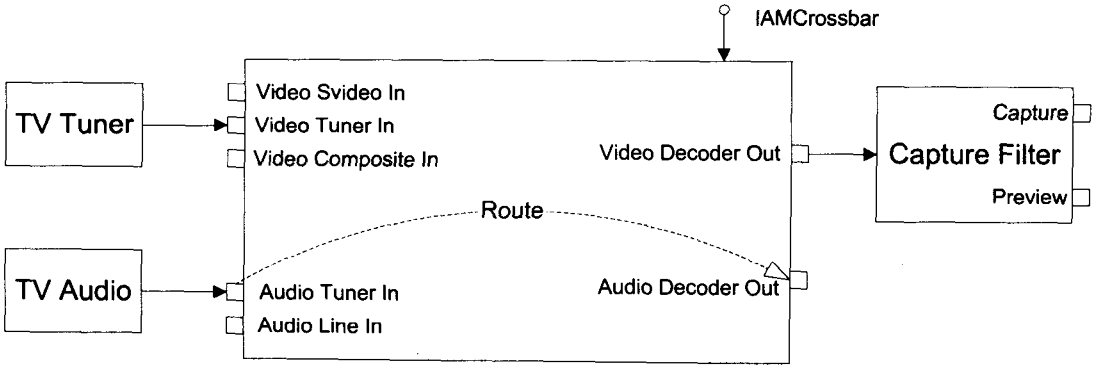

图 7.7　电视声音控制

简单地说，要听电视声音时，则要将 Crossbar 上的 Audio Decoder Out 输出 Pin 与 Audio Tuner In 输入 Pin 导通；需要静音时，要将 Audio Decoder Out 输出 Pin 与参数-1 导通。（上述导通操作使用接口方法 IAMCrossbar::Route 完成。）注意，在任何时候，都不用在 Filter Graph 中加入 Audio Renderer。

提示　大部分电视卡使用一个 Crossbar Filter，但也有的卡（如 ATI 的电视卡）使用两个 Crossbar。在第 2 种情况下，如果在一个 Crossbar 上找不到某个 Pin 的时候，应该尝试到另外一个 Crossbar 上去找。

上述电视声音控制的代码如下：

```
// 激活（或者关闭）电视声音，考虑到采集卡带有两个 Crossbar Filter 的情况
HRESULT ActivateAudio(ICaptureGraphBuilder2 *pBuild, IBaseFilter *pSrc,
  BOOL bActivate)
{
    // Search upstream for a crossbar
    IAMCrossbar *pXBar1 = NULL;
    HRESULT hr = pBuild->FindInterface(&LOOK_UPSTREAM_ONLY, NULL, pSrc,
        IID_IAMCrossbar, (void**)&pXBar1);
    if (SUCCEEDED(hr))
    {
        hr = ConnectAudio(pXBar1, bActivate);
        if (FAILED(hr))
        {
            // Look for another crossbar
            IBaseFilter *pF = NULL;
            hr = pXBar1->QueryInterface(IID_IBaseFilter, (void**)&pF);
            if (SUCCEEDED(hr))
            {
                // Search upstream for another one
                IAMCrossbar *pXBar2 = NULL;
                hr = pBuild->FindInterface(&LOOK_UPSTREAM_ONLY, NULL, pF,
                    IID_IAMCrossbar, (void**)&pXBar2);
                pF->Release();
                if (SUCCEEDED(hr))
```

```
                {
                    hr = ConnectAudio(pXBar2, bActivate);
                    pXBar2->Release();
                }
            }
        }
        pXBar1->Release();
    }
    return hr;
}
// 辅助函数：在 Crossbar Filter 上查找 Audio Decoder Out Pin 和 Audio Tuner In
// Pin，调用 IAMCrossbar::Route 函数进行电视声音的导通或者断开
HRESULT ConnectAudio(IAMCrossbar *pXBar, BOOL bActivate)
{
    // Look for the Audio Decoder output pin
    long i = 0;
    HRESULT hr = FindCrossbarPin(pXBar, PhysConn_Audio_AudioDecoder,
        PINDIR_OUTPUT, &i);
    if (SUCCEEDED(hr))
    {
        if (bActivate)  // Activate the audio
        {
            // Look for the Audio Tuner input pin
            long j = 0;
            hr = FindCrossbarPin(pXBar, PhysConn_Audio_Tuner,
                PINDIR_INPUT, &j);
            if (SUCCEEDED(hr))
            {
                return pXBar->Route(i, j);
            }
        }
        else  // Mute the audio
        {
            return pXBar->Route(i, -1);
        }
    }
    return E_FAIL;
}

// 辅助函数：在 Crossbar Filter 上查找指定类型的 Pin
HRESULT FindCrossbarPin(
    IAMCrossbar *pXBar,                    // Pointer to the crossbar
    PhysicalConnectorType PhysicalType,    // Pin type to match
    PIN_DIRECTION Dir,                     // Pin direction
    long *pIndex)         // Receives the index of the pin, if found
{
    BOOL bInput = (Dir == PINDIR_INPUT ? TRUE : FALSE);
    // Find out how many pins the crossbar has
    long cOut, cIn;
    HRESULT hr = pXBar->get_PinCounts(&cOut, &cIn);
```

```
    if (FAILED(hr)) return hr;
    // Enumerate pins and look for a matching pin.
    long count = (bInput ? cIn : cOut);
    for (long i = 0; i < count; i++)
    {
        long iRelated = 0;
        long ThisPhysicalType = 0;
        hr = pXBar->get_CrossbarPinInfo(bInput, i, &iRelated,
            &ThisPhysicalType);
        if (SUCCEEDED(hr) && ThisPhysicalType == PhysicalType)
        {
            // Found a match, return the index
            *pIndex = i;
            return S_OK;
        }
    }
    // Did not find a matching pin
    return E_FAIL;
}
```

当我们需要听电视声音时调用如下代码：

```
// Activate the audio
hr = ActivateAudio(pBuild, pCap, TRUE);
```

想将电视声音关掉时调用如下代码：

```
// Later, mute the audio
hr = ActivateAudio(pBuild, pCap, FALSE);
```

最后，我们来简单介绍一下 VBI 附加信息的控制问题。先看 VBI 输出 CC 信息的情况。因为不同的 Capture Filter 输出 VBI 信息的 Pin 有可能也是不同的。有的是通过 VBI Pin（Pin 的类型为 PIN_CATEGORY_VBI），有的则直接通过 CC Pin（Pin 的类型为 PIN_CATEGORY_CC）。可以简单使用 ICaptureGraphBuilder2::RenderStream 完成 CC 附加信息解码链路的构建，代码如下：

```
hr = pBuild->RenderStream(&PIN_CATEGORY_VBI, 0, pCap, 0, 0);
if (FAILED(hr))
{
    hr = pBuild->RenderStream(&PIN_CATEGORY_CC, 0, pCap, 0, 0);
}
```

构建的 Filter Graph 如图 7.8 所示。

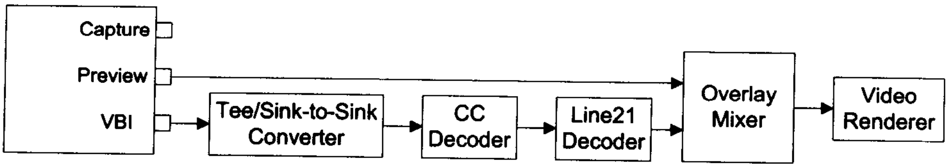

图 7.8　VBI 附加信息之 CC 控制

需要注意的是，有的 Capture Filter 可能还含有 VPVBI Pin。我们必须将这个 Pin 连接到一个 VBI Surface Allocator Filter，为它提供 Overlay 表面，如图 7.9 所示。（这一步操作会在 RenderStream 调用时根据需要由 Capture Graph Builder 自动加入。）

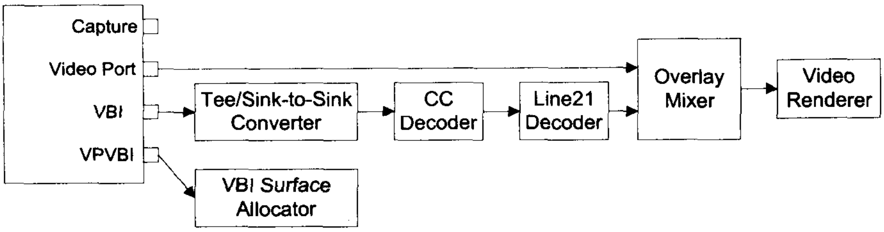

图 7.9　VPVBI Pin 的连接

我们再看 WST 的情况，它就没有那么幸运了！因为 ICaptureGraphBuilder2::RenderStream 不能直接支持 WST 的解码链路连接。我们需要自己加入各个 Filter，并依次完成连接，如图 7.10 所示。

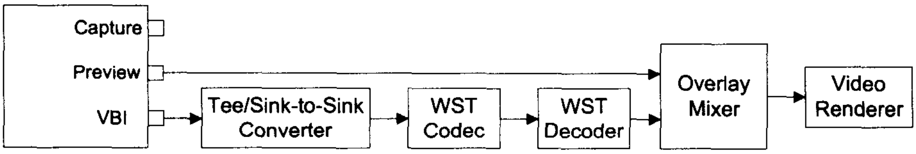

图 7.10　VBI 附加信息之 WST 控制

关于 CC 或 WST 信息的显示控制，我们可以通过 Line21 Decoder 上的 IAMLine21Decoder 接口或 WST Decoder 上的 IAMWstDecoder 接口来实现，方法类似，下面仅仅给出 CC 信息的控制代码：

```
// Use the FindInterface method to find the interface
IAMLine21Decoder *pLine21 = NULL;
hr = pBuild->FindInterface(
    &LOOK_DOWNSTREAM_ONLY, // Look downstream from pCap
    NULL,                  // No particular media type
    pCap,                  // Pointer to the capture filter
    IID_IAMLine21Decoder, (void**)&pLine21);
if (SUCCEEDED(hr))
{
    pLine21->SetServiceState(AM_L21_CCSTATE_Off);
    // (Use AM_L21_CCSTATE_On to enable.)
    pLine21->Release();
}
```

7.1.4　输入端子的选择

上面已经提到过，我们通过 Crossbar 来选择输入端子。那么，这在程序中是如何实现的呢？

首先当然要找到 Crossbar Filter。因为有的采集卡使用了两个 Crossbar Filtr，所以我们一般要做两次尝试。代码如下：

```
// Search upstream for a crossbar
IAMCrossbar *pXBar1 = NULL;
hr = pBuild->FindInterface(&LOOK_UPSTREAM_ONLY, NULL, pSrc,
        IID_IAMCrossbar, (void**)&pXBar1);
if (SUCCEEDED(hr))
{
    // Found one crossbar. Get its IBaseFilter interface
    IBaseFilter *pFilter = NULL;
    hr = pXBar1->QueryInterface(IID_IBaseFilter, (void**)&pFilter);
    if (SUCCEEDED(hr))
    {
        // Search upstream for another crossbar
        IAMCrossbar *pXBar2 = NULL;
        hr = pBuild->FindInterface(&LOOK_UPSTREAM_ONLY, NULL, pFilter,
                IID_IAMCrossbar, (void**)&pXBar2);
        pFilter->Release();

        if (SUCCEEDED(hr))
        {
            /* ... */
            pXBar2->Release();
        }
    }
    pXBar1->Release();
}
```

然后就是做“导通”工作。我们从图 7.7 看到，Crossbar Filter 最终只有一路视频和一路音频输出。所谓输入端子的选择，实际上是让 Crossbar Filter 的一个输入 Pin 与对应的一个输出 Pin 在 Filter 内部导通（通过调用接口方法 IAMCrossbar::Route）。比如要选择 S-Video 端子输入，我们就要把 Video SVideo In 与 Video Decoder Out 导通，同时把 Audio Line In 与 Audio Decoder Out 导通；如果选择 Video Tuner，则将 Video Tuner In 与 Video Decoder Out 导通，同时把 Audio Tuner In 与 Audio Decoder Out 导通；如果选择 Video Composite，则将 Video Composite In 与 Video Decoder Out 导通，同时把 Audio Line In 与 Audio Decoder Out 导通。

对于使用了两个 Crossbar 的采集卡，情况要稍微复杂一点，因为我们要做两个 Crossbar Filter 内部的导通工作，如图 7.11 所示。

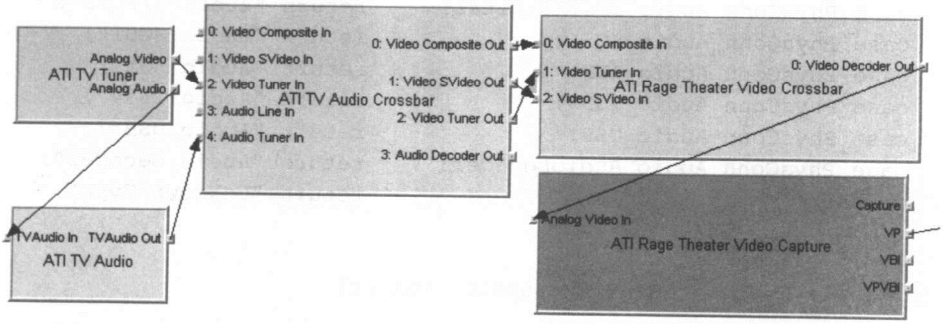

图 7.11　使用了两个 Crossbar 的 ATI 采集卡

如果选择 S-Video 端子输入，需要导通第 1 个 Crossbar 的 Video SVideo In 到 Video

SVideo Out，第 2 个 Crossbar 的 Video SVideo In 到 Video Decoder Out，同时导通 Audio Line In 到 Audio Decoder Out；如果选择 Video Tuner，需要导通第 1 个 Crossbar 的 Video Tuner In 到 Video Tuner Out，第 2 个 Crossbar 的 Video Tuner In 到 Video Decoder Out，同时导通 Audio Tuner In 到 Audio Decoder Out；如果选择 Video Composite，需要导通第 1 个 Crossbar 的 Video Composite In 到 Video Composite Out，第 2 个 Crossbar 的 Video Composite In 到 Video Decoder Out，同时导通 Audio Line In 到 Audio Decoder Out。

在实际的开发过程中，我们会用到 IAMCrossbar 的很多接口方法，比如 get_PinCounts 可以得到 Crossbar 上输入、输出 Pin 的数量，get_CrossbarPinInfo 可以得到 Pin 的类型（用一个枚举类型表示，有 PhysConn_Video_Tuner、PhysConn_Video_Composite、PhysConn_Video_SVideo 等）等。下面提供了这些接口方法的一段测试代码：

```
// Helper function to associate a name with the type
const char * GetPhysicalPinName(long lType)
{
    switch (lType)
    {
    case PhysConn_Video_Tuner:            return "Video Tuner";
    case PhysConn_Video_Composite:        return "Video Composite";
    case PhysConn_Video_SVideo:           return "S-Video";
    case PhysConn_Video_RGB:              return "Video RGB";
    case PhysConn_Video_YRYBY:            return "Video YRYBY";
    case PhysConn_Video_SerialDigital:    return "Video Serial Digital";
    case PhysConn_Video_ParallelDigital: return "Video Parallel Digital";
    case PhysConn_Video_SCSI:             return "Video SCSI";
    case PhysConn_Video_AUX:              return "Video AUX";
    case PhysConn_Video_1394:             return "Video 1394";
    case PhysConn_Video_USB:              return "Video USB";
    case PhysConn_Video_VideoDecoder:     return "Video Decoder";
    case PhysConn_Video_VideoEncoder:     return "Video Encoder";

    case PhysConn_Audio_Tuner:            return "Audio Tuner";
    case PhysConn_Audio_Line:             return "Audio Line";
    case PhysConn_Audio_Mic:              return "Audio Microphone";
    case PhysConn_Audio_AESDigital:       return "Audio AES/EBU Digital";
    case PhysConn_Audio_SPDIFDigital:     return "Audio S/PDIF";
    case PhysConn_Audio_SCSI:             return "Audio SCSI";
    case PhysConn_Audio_AUX:              return "Audio AUX";
    case PhysConn_Audio_1394:             return "Audio 1394";
    case PhysConn_Audio_USB:              return "Audio USB";
    case PhysConn_Audio_AudioDecoder:     return "Audio Decoder";
    default:                              return "Unknown Type";
    }
}
void DisplayCrossbarInfo(IAMCrossbar *pXBar)
{
    HRESULT hr;
    long cOutput = -1, cInput = -1;
    hr = pXBar->get_PinCounts(&cOutput, &cInput);
```

```
    for (long i = 0; i < cOutput; i++)
    {
        long lRelated = -1, lType = -1, lRouted = -1;

        hr = pXBar->get_CrossbarPinInfo(FALSE, i, &lRelated, &lType);
        hr = pXBar->get_IsRouted(i, &lRouted);

        printf("Output pin %d: %s\n", i, GetPhysicalPinName(lType));
        printf("\tRelated out: %d, Routed in: %d\n", lRelated, lRouted);
        printf("\tSwitching Matrix: ");

        for (long j = 0; j < cInput; j++)
        {
            hr = pXBar->CanRoute(i, j);
            printf("%d-%s", j, (S_OK == hr ? "Yes" : "No"));
        }
        printf("\n\n");
    }
    for (i = 0; i < cInput; i++)
    {
        long lRelated = -1, lType = -1;

        hr = pXBar->get_CrossbarPinInfo(TRUE, i, &lRelated, &lType);

        printf("Input pin %d - %s\n", i, GetPhysicalPinName(lType));
        printf("\tRelated in: %d\n", lRelated);
    }
}
```

如果我们在一个 Crossbar Filter 上调用上述 DisplayCrossbarInfo 函数，很有可能的输出代码如下：

```
Output pin 0: S-Video
    Related out: 2, Routed in: 0
    Switching Matrix: 0-Yes 1-No 2-No 3-No
Output pin 1 - Video Tuner
    Related out: 2, Routed in: 1
    Switching Matrix: 0-No 1-Yes 2-No 3-No
Output pin 2 - Audio decoder
    Related out: 1, Routed in: -1
    Switching Matrix: 0-No 1-No 2-Yes 3-Yes

Input pin 0 - S-Video
    Related in: 2
Input pin 1 - Video Tuner
    Related in: 3
Input pin 2 - Audio line
    Related in: 0
Input pin 3 - Audio tuner
    Related in: 1
```

提示

我们将在 7.2 节再次介绍 Crossbar 操作的更为通用的做法。

7.1.5 视频参数的设置

我们在预览或者采集之前，应对采集卡设置视频参数。这些参数包括采集图像的大小、帧率、输出格式，还有图像显示的亮度、对比度、色调、饱和度等。对于 VFW 卡，我们必须将驱动程序定制的设置对话框显示出来；而对于 WDM 卡，我们可以通过 IAMStreamConfig 接口和 IAMVideoProcAmp 接口编程来实现。

我们先来看看如何设置视频采集输出格式。首先是获得 Preview Pin 或者 Capture Pin 上的 IAMStreamConfig 接口，代码如下（使用 ICaptureGraphBuilder2::FindInterface）：

```
IAMStreamConfig *pConfig = NULL;
hr = pBuild->FindInterface(
    &PIN_CATEGORY_PREVIEW, // Preview pin
    0,    // Any media type
    pCap, // Pointer to the capture filter
    IID_IAMStreamConfig, (void**)&pConfig);
```

典型情况下，采集卡可以输出多种格式，如 16 位的 RGB 格式、32 位的 RGB 格式、YUYV 格式等。我们可以通过接口方法 IAMStreamConfig::GetNumberOfCapabilities 和 IAMStreamConfig::GetStreamCaps 进行枚举，代码如下：

```
int iCount = 0, iSize = 0;
hr = pConfig->GetNumberOfCapabilities(&iCount, &iSize);
// Check the size to make sure we pass in the correct structure
if (iSize == sizeof(VIDEO_STREAM_CONFIG_CAPS)
{
    // Use the video capabilities structure
    for (int iFormat = 0; iFormat < iCount; iFormat++)
    {
        VIDEO_STREAM_CONFIG_CAPS scc;
        AM_MEDIA_TYPE *pmtConfig;
        hr = pConfig->GetStreamCaps(iFormat, &pmtConfig, (BYTE*)&scc);
        if (SUCCEEDED(hr))
        {
            /* Examine the format, and possibly use it. */
            // Delete the media type when you are done.
            DeleteMediaType(pmtConfig);
        }
}
```

提示

GetNumberOfCapabilities 函数返回 Pin 上支持的输出格式的数量，描述输出格式的数据结构的大小（因为 IAMStreamConfig 接口即可用于视频格式的配置，也可用于音频格式的配置。视频使用 VIDEO_STREAM_CONFIG_CAPS 数据结构，音频使用 AUDIO_STREAM_CONFIG_CAPS 数据结构。我们需要通过这个大小来判断到底使用哪一个数据结构）。

在实际开发过程中，我们可以得到其中一个媒体类型，修改这种媒体类型描述的格式，比如图像的大小、帧率等，然后通过 IAMStreamConfig::SetFormat 设置回去，代码如下：

```
hr = pConfig->SetFormat(pmtConfig);
```

提示　当调用 IAMStreamConfig::SetFormat 时，如果 Pin 尚未连接，则在以后 Pin 连接时会使用设置的这种媒体类型；如果已经连接，则该 Pin 会用新的媒体类型进行重连。但是，这两种情况下设置的媒体类型都有可能不被下一级输入 Pin 所接受。

我们在修改媒体类型的时候，必须遵循 VIDEO_STREAM_CONFIG_CAPS 数据结构各成员的描述，比如 MinOutputSize 定义了图像输出的最小尺寸，MaxOutputSize 定义了最大尺寸，OutputGranularityX 定义了图像宽度改变的幅度，OutputGranularityY 定义了图像高度改变的幅度，MinFrameInterval 定义最小的单帧时间，MaxFrameInterval 定义了最大的单帧时间。举例来说，假设我们现在取得了一种如下的 24 位 RGB 格式的描述：

- MinOutputSize: 160×120
- MaxOutputSize: 320×240
- OutputGranularityX: 8 像素
- OutputGranularityY: 8 像素

于是，我们输出图像的宽在（160, 168, 176, ... 304, 312, 320）范围中的取值都是合法的，高在（120, 128, 136, ... 104, 112, 120）中的取值都是合法的。参考模型如图 7.12 所示。

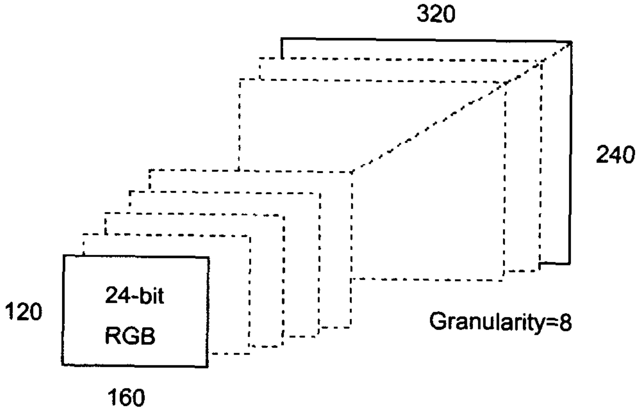

图 7.12　图像输出的合法尺寸

我们可以通过如下代码来改变输出图像的尺寸以及帧率：

```
if ((pmtConfig.majortype == MEDIATYPE_Video) &&
    (pmtConfig.subtype == MEDIASUBTYPE_RGB24) &&
    (pmtConfig.formattype == FORMAT_VideoInfo) &&
    (pmtConfig.cbFormat >= sizeof (VIDEOINFOHEADER)) &&
    (pmtConfig.pbFormat != NULL))
{
    VIDEOINFOHEADER *pVih = (VIDEOINFOHEADER*)pmtConfig.pbFormat;
```

```
    // pVih contains the detailed format information
    pVih->bmiHeader.biWidth  = 160;
    pVih->bmiHeader.biHeight = 120;
    pVih->bmiHeader.biSizeImage = DIBSIZE(pVih->bmiHeader);
    pVih->AvgTimePerFrame = 400000;
}
```

提示

对于一些 PCI 接口的采集卡，我们在设置视频输出格式时应该考虑到 PCI 总线的数据传送能力。如果设置的图像尺寸太大或者帧率太高，将导致 Preview Pin 和 Capture Pin 加起来的数据流量超过总线限制，结果将是 Preview Pin 或 Capture Pin 只能成功连接一个，甚至两个都不能连接成功。

接下去，我们再来看一下图像显示参数的设置。首先我们要在采集卡 Filter 上获得 IAMVideoProcAmp 接口，然后通过这个接口的 Get 方法和 Set 方法取得、设置各个参数值。参数类型定义采用一个枚举类型结构，代码如下：

```
typedef enum tagVideoProcAmpProperty
{
    VideoProcAmp_Brightness,   // 亮度
    VideoProcAmp_Contrast,     // 对比度
    VideoProcAmp_Hue,          // 色调
    VideoProcAmp_Saturation,   // 饱和度
    VideoProcAmp_Sharpness,
    VideoProcAmp_Gamma,
    VideoProcAmp_ColorEnable,
    VideoProcAmp_WhiteBalance,
    VideoProcAmp_BacklightCompensation,
    VideoProcAmp_Gain
} VideoProcAmpProperty;
```

以亮度为例，我们给出了一个实现通过一个滚动控件显示当前亮度值的代码：

```
HWND hTrackbar; // Handle to the trackbar control
// Initialize hTrackbar (not shown)
// Query the capture filter for the IAMVideoProcAmp interface
IAMVideoProcAmp *pProcAmp = 0;
hr = pCap->QueryInterface(IID_IAMVideoProcAmp, (void**)&pProcAmp);
if (FAILED(hr))
{
    // The device does not support IAMVideoProcAmp, so disable the control
    EnableWindow(hTrackbar, FALSE);
}
else
{
    long Min, Max, Step, Default, Flags, Val;
    // Get the range and default value
    hr = m_pProcAmp->GetRange(VideoProcAmp_Brightness, &Min, &Max, &Step,
        &Default, &Flags);
    if (SUCCEEDED(hr))
```

```
    {
        // Get the current value
        hr = m_pProcAmp->Get(VideoProcAmp_Brightness, &Val, &Flags);
    }
    if (SUCCEEDED(hr))

        // Set the trackbar range and position
        SendMessage(hTrackbar, TBM_SETRANGE, TRUE, MAKELONG(Min, Max));
        SendMessage(hTrackbar, TBM_SETPOS, TRUE, Val);
        EnableWindow(hTrackbar, TRUE);
    }
    else
    {
        // This property is not supported, so disable the control
        EnableWindow(hTrackbar, FALSE);
    }
}
```

7.1.6 热插拔的支持

一些采用支持热插拔接口（如 USB、1394 等）的采集卡，在工作的时候可能会突然脱离系统，然后可能又马上重新接入，或者也有可能在应用程序启动之后才接入一个新的设备。在这些情况下，我们应该如何分别做出响应呢？

当一个正在使用的采集卡脱离系统时，Filter Graph Manager 会向应用程序发出一个 EC_DEVICE_LOST 事件；而当这个设备重新接入系统时，Filter Graph Manager 会再次发出一个 EC_DEVICE_LOST 事件。这两次事件通过事件的第 2 个参数 lParam2 来区分：0 表示设备脱离系统，1 表示设备又重新接入系统。但是实际上，设备脱离后再重新接入，采集卡 Filter 先前的状态不会保持下来，因此我们必须重建 Filter Graph。

另外一种情况是，当我们的应用程序正在运行时，有一个全新的设备接入系统，但 DirectShow 在这种情况下并不会发送 EC_DEVICE_LOST 事件。我们必须依赖窗口消息 WM_DEVICECHANGE 来做出正确的响应。实现步骤如下：

（1）从 CFrameWnd 类派生一个子类 CDeviceController（CFrameWnd 类能够接收到系统发出的 WM_DEVICECHANGE 消息），代码如下：

```
class CDeviceController : public CFrameWnd
```

（2）在 CDeviceController 类中定义 WM_DEVICECHANGE 的消息响应函数，代码如下：

```
BEGIN_MESSAGE_MAP(CDeviceController, CFrameWnd)
    //{{AFX_MSG_MAP(CDeviceDispatcher)
    ON_WM_CREATE()
    ON_WM_CLOSE()
    ON_WM_DEVICECHANGE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
//{{AFX_MSG(CDeviceController)
```

```
    afx_msg void OnClose();
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg BOOL OnDeviceChange( UINT nEventType, DWORD dwData );
//}}AFX_MSG
```

（3）在创建窗口的时候，把我们的窗口注册为设备消息接收器，代码如下：

```
int CDeviceController::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    int createResult = CFrameWnd::OnCreate(lpCreateStruct);
    if (createResult != -1)
    {
          Register();
    }
    return createResult;
}
```

其中，Register 是我们在 CDeviceController 类中实现的注册函数，我们需要从 user32.dll 中导出注册函数（以及注销函数）。我们定义一些常量及数据类型，代码如下：

```
#if (WINVER < 0x0500)
#define DBT_DEVTYP_DEVICEINTERFACE 0x00000005  // device interface class
#define DBT_DEVTYP_HANDLE          0x00000006  // file system handle
#define DEVICE_NOTIFY_WINDOW_HANDLE     0x00000000
typedef  PVOID            HDEVNOTIFY;
#endif
extern "C"
{
    typedef BOOL (WINAPI *PUnregisterDeviceNotification)(IN HDEVNOTIFY
Handle);
    typedef HDEVNOTIFY (WINAPI *PRegisterDeviceNotification)(IN HANDLE
hRecipient,IN LPVOID NotificationFilter,IN DWORD Flags);
}
#if (WINVER < 0x0500)
typedef struct _DEV_BROADCAST_DEVICEINTERFACE {
    DWORD          dbcc_size;
    DWORD          dbcc_devicetype;
    DWORD          dbcc_reserved;
    GUID           dbcc_classguid;
    char           dbcc_name[1];
} DEV_BROADCAST_DEVICEINTERFACE, *PDEV_BROADCAST_DEVICEINTERFACE;
#endif
```

我们还需在类 CDeviceController 中定义如下几个成员变量：

```
HDEVNOTIFY                      mNotifyHandle;
PRegisterDeviceNotification     mRegisterDeviceNotification;
PUnregisterDeviceNotification   mUnregisterDeviceNotification;
```

注册函数 Register 实现的代码如下：

```
void CDeviceController::Register(void)
{
```

```
    DEV_BROADCAST_DEVICEINTERFACE filterData;
    ZeroMemory(&filterData, sizeof(DEV_BROADCAST_DEVICEINTERFACE));
    filterData.dbcc_size = sizeof(DEV_BROADCAST_DEVICEINTERFACE);
    filterData.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
    filterData.dbcc_classguid = AM_KSCATEGORY_CAPTURE;
    mUnregisterDeviceNotification = NULL;
    mRegisterDeviceNotification = NULL;
    // dynload device removal APIs
    {
          HMODULE hmodUser = GetModuleHandle(TEXT("user32.dll"));
          ASSERT(hmodUser);      // we link to user32
         mUnregisterDeviceNotification = (PUnregisterDeviceNotification)

    GetProcAddress(hmodUser, "UnregisterDeviceNotification");
          // m_pRegisterDeviceNotification is prototyped differently in
          // unicode
          mRegisterDeviceNotification = (PRegisterDeviceNotification)
                                         GetProcAddress(hmodUser,
  #ifdef UNICODE
                                      "RegisterDeviceNotificationW"
  #else
                                      "RegisterDeviceNotificationA"
  #endif
                                      );
          // failures expected on older platforms
          ASSERT(mRegisterDeviceNotification &&
  mUnregisterDeviceNotification ||
                     !mRegisterDeviceNotification
&& !mUnregisterDeviceNotification);
    }

    mNotifyHandle = NULL;
    if (mRegisterDeviceNotification)
    {
            mNotifyHandle = mRegisterDeviceNotification(this->
 GetSafeHwnd(), &filterData, DEVICE_NOTIFY_WINDOW_HANDLE);
          ASSERT(mNotifyHandle != NULL);
    }
  }
```

（4）实现 **OnDeviceChange** 函数（动态维护系统设备列表），代码如下：

```
  BOOL CDeviceController::OnDeviceChange( UINT nEventType, DWORD dwData )
  {
    // We are interested in only device arrival & removal events
    if (DBT_DEVICEARRIVAL != nEventType && DBT_DEVICEREMOVECOMPLETE !=
nEventType)
          return CWnd::OnDeviceChange (nEventType, dwData);

    // 这里我们可以维护一个系统设备列表
    // …
```

```
    return CWnd::OnDeviceChange (nEventType, dwData);
}
```

（5）在 OnClose 函数中把我们的窗口从设备消息接收器列表中注销，代码如下：

```
void CDeviceController::OnClose()
{
    Unregister();
    CFrameWnd::OnClose();
}
void CDeviceController::Unregister(void)
{
    if (mNotifyHandle != NULL)
    {
            ASSERT(mUnregisterDeviceNotification);
            mUnregisterDeviceNotification(mNotifyHandle);
            mNotifyHandle = NULL;
    }
}
```

7.2 实 例 解 剖

DirectShow SDK 提供了一个视频采集的例子，这个例子位于 SDK 安装目录的 Samples\C++\DirectShow\Capture\AMCap 下。如果我们正在开发音、视频采集应用，AMCap 将是最值得参考和学习的例子。下面，我们就来介绍这个例子。

7.2.1 实现的功能

如图 7.13 所示是 AMCap 例子的程序界面。这个应用程序完成的主要功能有：

- 枚举系统中所有的音、视频采集设备，并生成列表；
- 对指定的采集设备进行视频内容的预览，并支持采集 AVI 文件；
- 支持电视 VBI 附加信息 CC 的显示和采集；
- 支持 MPEG2 流的输出；
- 支持设备的热插拔。

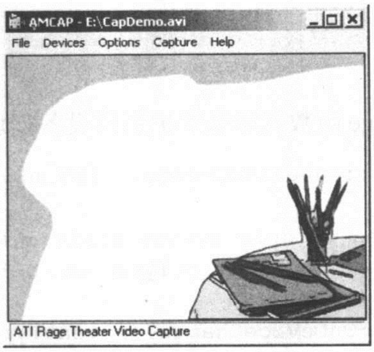

图 7.13　AMCap 例子的程序界面

7.2.2 实现要点

通过 AMCap 例子的学习，我们将掌握如下知识点：

- Win32 应用程序的编程；
- 使用 Capture Graph Builder 辅助构建采集应用的 Filter Graph；
- 操作 Crossbar 进行视频输入端子的选择；
- 设置音、视频采集输出的格式；
- 支持采集设备的热插拔；
- 显示与各个采集相关的 Filter 和 Pin 的属性页；
- 注册 Filter Graph 到 ROT，以便 GraphEdit 查看。

Win32 应用程序的编程

AMCap 是一个标准的 Win32 应用程序。程序执行的主要流程是：WinMain | AppInit | AddDevicesToMenu | ChooseDevices | BuildPreviewGraph | 消息循环……

通过 AMCap 例子的学习，我们能够学到很多基本的 Win32 应用程序编程知识，包括窗口的创建过程、菜单项操作、消息循环与响应等。相对于 DirectShow 这一主题，这些知识当然只能算是“副产品”。

使用 Capture Graph Builder 辅助构建采集应用的 Filter Graph

Filter Graph 的构建主要包括在 BuildCaptureGraph 和 BuildPreviewGraph 两个函数的实现中。我们注意到，gcap.pBuilder 实际上是一个 IsampleCaptureGraphBuilder 类型的指针，它在 MakeBuilder 函数中创建，代码如下：

```
BOOL MakeBuilder()
{
    // we have one already
    if(gcap.pBuilder)
        return TRUE;

    gcap.pBuilder = new ISampleCaptureGraphBuilder( );
    if( NULL == gcap.pBuilder )
    {
        return FALSE;
    }

    return TRUE;
}
```

那它为什么能够直接调用 ICaptureGraphBuilder2 的接口方法呢？

我们从 SampleCGB.cpp 文件的实现中不难发现，其实类 ISampleCaptureGraphBuilder 内部封装了一个 Capture Graph Builder 的组件。这个组件在 ISampleCaptureGraphBuilder 的类初始化函数中即被创建，代码如下：

```
ISampleCaptureGraphBuilder()
{
```

```
        AudPID_ = 0xC0;
        VidPID_ = 0xE0;
        HRESULT hr =
graphBuilder2_.CoCreateInstance( CLSID_CaptureGraphBuilder2 );
        ASSERT( S_OK == hr );
}
```

我们在外部调用 gcap.pBuilder 上的接口方法，其实大部分都是转给了内部组件对应的接口方法。使用 ISampleCaptureGraphBuilder 扩展 ICaptureGraphBuilder2 接口的目的，主要是为了支持 MPEG2 流的应用。

在 BuildCaptureGraph 或 BuildPreviewGraph 函数的实现中，都是使用 gcap.pBuilder ->RenderStream 调用，来完成指定 Pin 的后续 Filter 链路的构建，同时通过 IVideoWindow 接口进行视频窗口的一些设置、视频输出格式的设置等。另外，还有一个很有用的函数——TearDownGraph——的“精妙”实现。

操作 Crossbar 进行视频输入端子的选择

因为有的采集卡使用一个 Crossbar Filter，有的使用两个。这在实际写程序的时候有点麻烦。于是，AMCap 例子实现了一个 CCrossbar 类专门负责处理 Crossbar 问题。即使在我们自己开发应用程序的时候，也可以直接将 crossbar.cpp 和 crossbar.h 文件加入到我们的项目中使用。

对于 CCrossbar 类的理解，关键在于递归函数 BuildRoutingList 的实现上。该函数建立了一个 Crossbar Filter 的输入和输出 Pin 之间“导通”的映射表。当我们实际选择某个输入端子的时候，只需要根据这个表调用 IAMCrossbar::Route 就行了。

CCrossbar 类的使用方法，我们可以参见 InitCapFilters 函数，代码如下：

```
// Use the crossbar class to help us sort out all the possible video
// inputs.The class needs to be given the capture filters ANALOGVIDEO
// input pin
{
        IPin         *pP = 0;
        IEnumPins    *pins=0;
        ULONG         n;
        PIN_INFO      pinInfo;
        BOOL          Found = FALSE;
        IKsPropertySet *pKs=0;
        GUID guid;
        DWORD dw;
        BOOL fMatch = FALSE;
        gcap.pCrossbar = NULL;
        if(SUCCEEDED(gcap.pVCap->EnumPins(&pins)))
        {
            while(!Found && (S_OK == pins->Next(1, &pP, &n)))
            {
                if(S_OK == pP->QueryPinInfo(&pinInfo))
                {
                    if(pinInfo.dir == PINDIR_INPUT)
                    {
```

```
                        // is this pin an ANALOGVIDEOIN input pin?
                        if(pP->QueryInterface(IID_IKsPropertySet,
                            (void **)&pKs) == S_OK)
                        {
                            if(pKs->Get(AMPROPSETID_Pin,
                                AMPROPERTY_PIN_CATEGORY, NULL, 0,
                                &guid, sizeof(GUID), &dw) == S_OK)
                            {
                                if(guid == PIN_CATEGORY_ANALOGVIDEOIN)
                                    fMatch = TRUE;
                            }
                            pKs->Release();
                        }
                        if(fMatch)
                        {
                            HRESULT hrCreate=S_OK;
                            gcap.pCrossbar = new CCrossbar(pP, &hrCreate);
                            if (!gcap.pCrossbar || FAILED(hrCreate))
                                break;
                            hr = gcap.pCrossbar->GetInputCount(&gcap.
                                NumberOfVideoInputs);
                            Found = TRUE;
                        }
                    }
                    pinInfo.pFilter->Release();
                }
                pP->Release();
            }
            pins->Release();
        }
    }
```

我们首先要枚举采集卡 Filter 的所有 Pin，查询得到一个称为 Analog Video In 的输入 Pin，然后将这个 Pin 作为 CCrossbar 类的一个参数，创建一个 CCrossbar 对象实例。之后，就只要调用 CCrossbar 对象上提供的函数接口就可以控制输入端子，比如 GetInputCount 得到输入端子的数量，GetInputType 得到输入端子的类型，GetInputName 得到输入端子的名字，SetInputIndex 设置输入端子等。

音、视频采集输出格式的设置

AMCap 例子演示了如何修改视频采集的帧率，我们在 BuildCaptureGraph 或 BuildPreviewGraph 函数中都能看到这样的实现。

```
// 得到当前 Capture Pin 使用的媒体类型，然后仅仅改变帧率后再设置回去
if(gcap.pVSC && gcap.fUseFrameRate)
{
        hr = gcap.pVSC->GetFormat(&pmt);
        // DV capture does not use a VIDEOINFOHEADER
        if(hr == NOERROR)
        {
            if(pmt->formattype == FORMAT_VideoInfo)
```

```
            {
                VIDEOINFOHEADER *pvi = (VIDEOINFOHEADER *)pmt->pbFormat;
                pvi->AvgTimePerFrame = (LONGLONG)(10000000 /
gcap.FrameRate);
                hr = gcap.pVSC->SetFormat(pmt);
                if(hr != NOERROR)
                    ErrMsg(TEXT("%x: Cannot set frame rate for preview"),
hr);
            }
            DeleteMediaType(pmt);
        }
    }
```

另外，还可以在 ChooseAudioFormat 函数中，学习到音频采集输出格式的设置方法。注意该函数中使用了 acmMetrics、acmFormatChoose 等 API 函数。

采集设备热插拔的支持

AMCap 例子定义了一个窗口消息 WM_FGNOTIFY，用于通知 Filter Graph Manager 发出的事件。WM_FGNOTIFY 的消息响应中有 EC_DEVICE_LOST 事件的处理，代码如下：

```
    else if(event == EC_DEVICE_LOST)
    {
        // Check if we have lost a capture filter being used
        // lParam2 of EC_DEVICE_LOST event == 1 indicates device added
        //                                  == 0 indicates device removed
        if(l2 == 0)
        {
            IBaseFilter *pf;
            IUnknown *punk = (IUnknown *) (LONG_PTR) l1;
            if(S_OK == punk->QueryInterface(IID_IBaseFilter, (void **) &pf))
            {
                if(::IsEqualObject(gcap.pVCap, pf))
                {
                    pf->Release();
                    bAbort = FALSE;
                    StopCapture();
                    TCHAR szError[100];
                    _tcscpy(szError,
                        TEXT("Stopping Capture (Device Lost). Select New
Capture Device\0"));
                    ErrMsg(szError);
                    break;
                }
                pf->Release();
            }
        }
    }
```

当接收到设备脱离的通知后，AMCap 判断这个设备是否就是当前正在使用的那个。

如果是，则调用 StopCapture 停止当前的采集进程。

WM_DEVICECHANGE 的消息响应代码如下：

```
case WM_DEVICECHANGE:
    // We are interested in only device arrival & removal events
    if(DBT_DEVICEARRIVAL != wParam && DBT_DEVICEREMOVECOMPLETE != wParam)
        break;
    PDEV_BROADCAST_HDR pdbh = (PDEV_BROADCAST_HDR) lParam;
    if(pdbh->dbch_devicetype != DBT_DEVTYP_DEVICEINTERFACE)
    {
        break;
    }
    PDEV_BROADCAST_DEVICEINTERFACE pdbi =
(PDEV_BROADCAST_DEVICEINTERFACE) lParam;
    // Check for capture devices
    if(pdbi->dbcc_classguid != AM_KSCATEGORY_CAPTURE)
    {
        break;
    }
    // Check for device arrival/removal
    if(DBT_DEVICEARRIVAL == wParam || DBT_DEVICEREMOVECOMPLETE == wParam)
    {
        gcap.fDeviceMenuPopulated = false;
    }
    break;
```

从代码中可以看到，当系统中的采集设备脱离或者接入后，gcap.fDeviceMenuPopulated 被置为 false。于是以后调用 AddDevicesToMenu 时，会进行系统设备列表的刷新（生成菜单项）。

与各个采集相关的 Filter 和 Pin 属性页的显示

只有 VFW 卡才有 3 个设置对话框，可以直接使用 IAMVfwCaptureDialogs 接口将它们显示出来。我们可以在 AMCap 例子的 video format dialog 部分的消息响应中，看到 gcap.pDlg->ShowDialog 的函数调用，参数分别为 VfwCaptureDialog_Format、VfwCaptureDialog_Source、VfwCaptureDialog_Display，表示 3 种不同类型的设置对话框。

WDM 卡可以通过 OleCreatePropertyFrame 函数显示 Filter 或 Pin 上的属性页。这部分内容同样在 AMCap 例子的 video format dialog 部分的消息响应中。以显示视频采集卡 Filter 的属性页为例，代码如下：

```
else if(id - MENU_DIALOG0 == gcap.iVCapDialogPos)
{
    ISpecifyPropertyPages *pSpec;
    CAUUID cauuid;
    hr = gcap.pVCap->QueryInterface(IID_ISpecifyPropertyPages,
        (void **)&pSpec);
    if(hr == S_OK)
    {
        hr = pSpec->GetPages(&cauuid);
```

```
        hr = OleCreatePropertyFrame(ghwndApp, 30, 30, NULL, 1,
            (IUnknown **)&gcap.pVCap, cauuid.cElems,
            (GUID *)cauuid.pElems, 0, 0, NULL);

        CoTaskMemFree(cauuid.pElems);
        pSpec->Release();
    }
}
```

注册 Filter Graph 到 ROT，以便于 GraphEdit 查看

AMCap 例子实现了两个函数：AddGraphToRot 和 RemoveGraphFromRot，分别完成 Filter Graph 到 ROT 的注册和注销。注意，在 BuildCaptureGraph 或 BuildPreviewGraph 函数中都调用了 AddGraphToRot，代码如下：

```
#ifdef REGISTER_FILTERGRAPH
    hr = AddGraphToRot(gcap.pFg, &g_dwGraphRegister);
    if(FAILED(hr))
    {
        ErrMsg(TEXT("Failed to register filter graph with ROT!
hr=0x%x"), hr);
        g_dwGraphRegister = 0;
    }
#endif
```

而在 TearDownGraph 函数中调用了 RemoveGraphFromRot，代码如下：

```
#ifdef REGISTER_FILTERGRAPH
    // Remove filter graph from the running object table
    if(g_dwGraphRegister)
    {
        RemoveGraphFromRot(g_dwGraphRegister);
        g_dwGraphRegister = 0;
    }
#endif
```

第 8 章　数码摄像机的支持

8.1　应 用 分 析

数码摄像机（DV 机）的应用其实也属于音、视频采集的范畴，比如通过摄像头实时采集、磁带内容抓取等。

DirectShow 同样是以包装 Filter 的方式提供对 DV 机的支持，这个 Filter 的名字（Friendly Name）总是显示为 Microsoft DV Camera and VCR。如果我们已经将 DV 机打开，并且也已经通过一根 1394 线接入了电脑，运行 GraphEdit 并选择插入 Filter 时，在 Video Capture Sources 目录下就能找到这个 Filter。然后，在 Filter Graph 中加入这个 Filter，如图 8.1 所示。

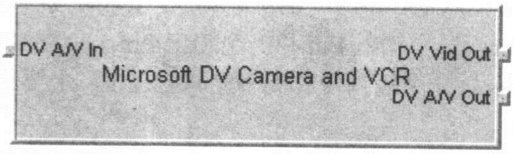

图 8.1　代表 DV 机的 Filter

我们看到，DV 机的 Filter 有一个输入 Pin 和两个输出 Pin。输入 Pin 为 DV A/V In，仅当我们将外部数据通过 1394 连线保存到 DV 机的磁带上时使用。一个输出 Pin 为 DV Vid Out，仅输出视频数据；另外一个为 DV A/V Out，输出音、视频交叉的 DV 数据。

当我们的 DV 机工作在采集模式（摄像头实时采集或磁带内容抓取）时，DV 机的 Filter 类似于一个 Source Filter（此时那个输入 Pin 是不用连接的），如图 8.2 所示。

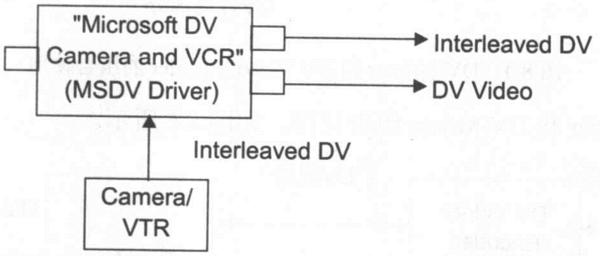

图 8.2　DV 机的采集模式

当我们的 DV 机工作在录像模式时，DV 机的 Filter 类似于一个 Renderer Filter（此时两个输入 Pin 是不用连接的），如图 8.3 所示。

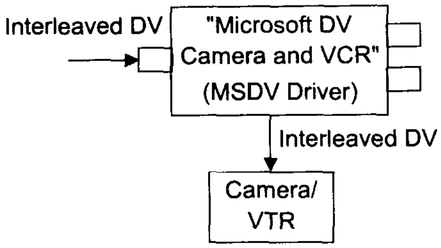

图 8.3　DV 机的录像模式

DV 格式的数据，视频采用帧内压缩，音频仍然为 PCM 格式，音、视频采用交叉的方式组成一帧。一般 PAL 制的图像大小为 720×576，单帧为 144 000B，NTSC 制的图像大小为 720×480，单帧为 120 000B。所以将 DV 数据直接保存为文件，对于磁盘容量的需求还是很大的。

除了 DV 机 Filter 外，DirectShow 还提供了一系列相关的 Filter 来处理 DV 数据：

- DV Splitter：一个输入 Pin 和两个输出 Pin，主要用于 DV 数据的音、视频分离；
- DV Video Decoder：一个输入 Pin 和一个输出 Pin，负责 DV 的视频解码；
- DV Video Encoder：一个输入 Pin 和一个输出 Pin，主要将视频数据编码成 DV 格式；
- DV Muxer：两个输入 Pin 和一个输出 Pin，主要将独立的音、视频数据合成为一个 DV 流。

一般 DV Splitter 和 DV Video Decoder 会组合使用，如图 8.4 所示。

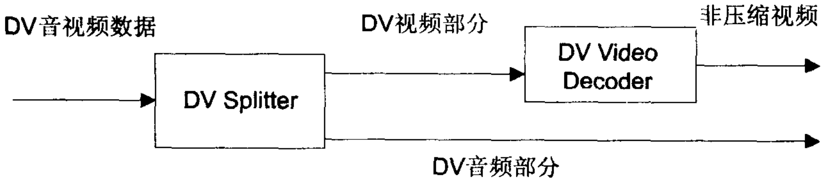

图 8.4　DV Splitter 和 DV Video Decoder 的组合使用

DV Video Encoder 和 DV Muxer 组合使用，如图 8.5 所示。

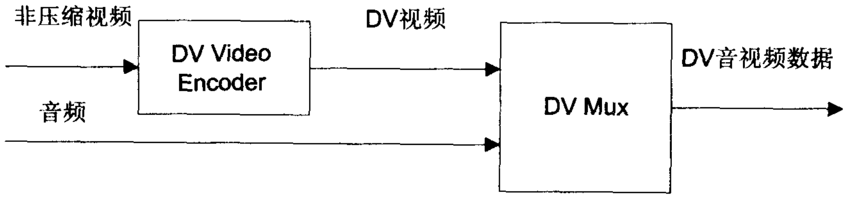

图 8.5　DV Video Encoder 和 DV Muxer 的组合使用

DV Splitter 和 DV Mux 用于 Type-1 类型的 DV 数据与 Type-2 类型的 DV 数据之间的相互转换，如图 8.6 所示。

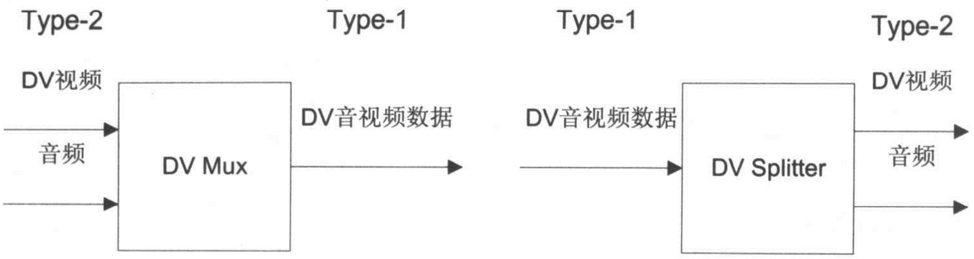

图 8.6　Type-1 类型与 Type-2 类型的 DV 格式互转

提示　一般 DV 数据可以直接保存到 AVI 文件中。保存时有两种格式：Type-1 和 Type-2。前者是将 DV 数据以原有的音、视频交叉格式直接保存，后者是将 DV 数据的音、视频分离后再保存。实际上，Type-2 的视频数据仍然带有音频的冗余数据，它只是简单地被标为视频而将音频数据忽略而已。Type-2 主要是为了保持 VFW 的向后兼容性的。而在一些实时采集应用中，推荐使用 Type-1，这样可以加快 CPU 速度。

8.1.1　磁带的播放

如何实现对 DV 机中的磁带的播放？我们在 GraphEdit 中，可以构建如图 8.7 所示的 Filter Graph。

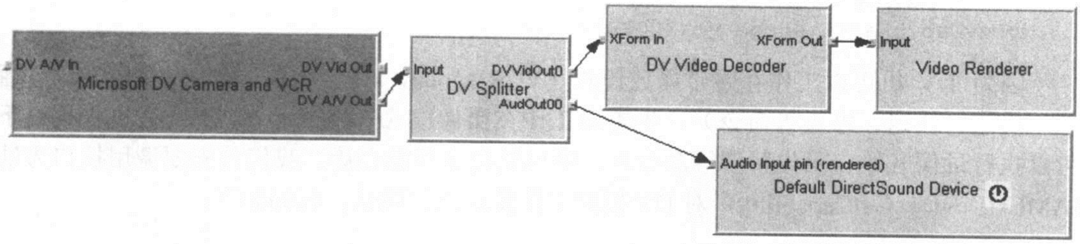

图 8.7　DV 机磁带播放 Filter Graph

运行 Filter Graph，就可以通过 DV 机 Filter 的属性页进行控制，如图 8.8 所示。

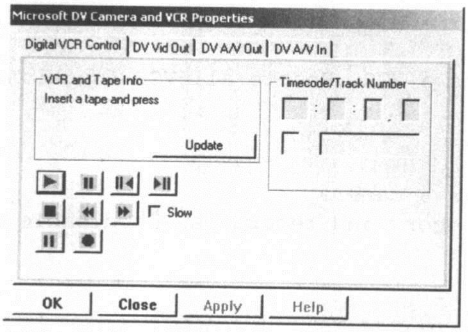

图 8.8　磁带播放时 DV 机 Filter 的属性页

那么，我们如何在程序中实现对 DV 机中的磁带的播放？很简单，这些功能都可以通过 DV 机 Filter 上的 IAMExtDevice、IAMExtTransport 和 IAMTimecodeReader 3 个接口来完成。

提示 如果要使用这些接口，应该在应用程序中包含 XPrtDefs.h 头文件。

获得上述控制接口的代码如下：

```
struct _MyDevCap
{
    IAMExtDevice        *pDevice;
    IAMExtTransport     *pTransport;
    IAMTimecodeReader   *pTimecode;
    BOOL                 bHasDevice;
    BOOL                 bHasTransport;
    BOOL                 bHasTimecode;
} MyDevCap;
HRESULT hr;
IBaseFilter *pDVCam;  // Pointer to the capture filter
// Create an instance of the capture filter (not shown)
hr = pDVCam->QueryInterface(IID_IAMExtDevice, (void
     **)&MyDevCap.pDevice);
MyDevCap.bHasDevice = (SUCCEEDED(hr));
hr = pDVCam->QueryInterface(IID_IAMExtTransport, (void
     **)&MyDevCap.pTransport);
MyDevCap.bHasTransport = (SUCCEEDED(hr));
hr = pDVCam->QueryInterface(IID_IAMTimecodeReader, (void
     **)&MyDevCap.pTimecode);
MyDevCap.bHasTimecode = (SUCCEEDED(hr));
```

因为 DV 机可以工作在磁带播放模式（VCR Mode，在这种模式下可以执行播放、暂停、快进、快退、停止等命令），也可以工作在摄像模式（Camera Mode，在这种模式下可以执行录像开始、录像暂停等命令），所以获得这些接口后，我们首先要调用接口方法 IAMExtDevice::GetCapability，对 DV 机的工作模式进行确认，代码如下：

```
if (MyDevCap.bHasDevice)
{
    LONG lDeviceType = 0;
    MyDevCap.pDevice->GetCapability(ED_DEVCAP_DEVICE_TYPE,  &lDeviceType,
0);
    if (lDeviceType == ED_DEVTYPE_VCR)
    {
        // Device is a VCR. Enable all VCR functions
    }
    else
    {
        // Device is a camera
        // Enable record and record-pause; disable other functions
    }
}
```

另外，DV 机有可能是 PAL 制式的，也有可能是 NTSC 制式的，那么我们在程序中如何识别呢？调用接口方法 IAMExtTransport::GetTransportBasicParameters，并且将第 1 个参

数指定为 ED_TRANSBASIC_INPUT_SIGNAL 即可，代码如下：

```
LONG Mode, AvgTimePerFrame;
hr = MyDevCap.pTransport->GetTransportBasicParameters(
        ED_TRANSBASIC_INPUT_SIGNAL, &Mode, NULL);
if (SUCCEEDED(hr))
{
    switch (Mode)
    {
    case ED_TRANSBASIC_SIGNAL_525_60_SD: // NTSC SD
        AvgTimePerFrame = 33;  // 33 msec (29.97 FPS)
        break;
    case ED_TRANSBASIC_SIGNAL_525_60_SDL: // NTSC SDL
        AvgTimePerFrame = 33;
        break;
    case ED_TRANSBASIC_SIGNAL_625_50_SD: // PAL SD
        AvgTimePerFrame = 40;  // 40 msec (25 FPS)
        break;
    case ED_TRANSBASIC_SIGNAL_625_50_SDL: // PAL SDL
        AvgTimePerFrame = 40;
        break;
    default:
        // Unknown type
        AvgTimePerFrame = 33; // Default
        break;
    }
}
```

下面介绍对 DV 机播放的控制（通过 IAMExtTransport::put_Mode），以及 DV 机状态的刷新（通过 IAMExtTransport::get_Mode 或 IAMExtTransport::GetStatus）。

DV 机常见的状态见表 8.1。

表 8.1　DV 机常见状态

状态值	状态描述
ED_MODE_PLAY	播放
ED_MODE_STOP	停止
ED_MODE_FREEZE	（播放）暂停
ED_MODE_FF	快进
ED_MODE_REW	快退
ED_MODE_RECORD	录像
ED_MODE_RECORD_FREEZE	（录像）暂停

提示

在 DV 机 Filter 上实现 IAMExtDevice、IAMExtTransport 和 IAMTimecodeReader 的方法是，将各接口方法转化为对应的指令发给驱动程序，当驱动程序执行指令后，DV 机 Filter 检查它的返回值，然后以 HRESULT 的形式再返回给上层调用者。因为 DV 机是一种物理设备，完成一个指令需要一定的时间，这就要求我们的应用程序以一种“异步”的方式获取 DV 机当前真正的状态，进而准确地刷新界面。

实现 DV 机状态的设置的代码如下：

```
MyDevCap.pTransport->put_Mode(ED_MODE_STOP);
```

获得 DV 机当前状态的代码如下：

```
LONG State;
hr = MyDevCap.pTransport->get_Mode(&State);
if (SUCCEEDED(hr))
{
    switch (State)
    {
        case ED_MODE_PLAY:   // 播放
              break;
        case ED_MODE_STOP:   // 停止
              break;
        case ED_MODE_FREEZE: // 暂停
                  break;
    }
}
```

下面介绍一种使用独立的辅助线程（内部调用 IAMExtTransport::GetStatus）刷新 DV 机状态的方法，步骤如下：

（1）创建几个同步事件对象，并创建一个辅助线程，代码如下：

```
CRITICAL_SECTION csIssueCmd;
HANDLE   hThreadEndEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
// 控制线程退出的事件
HANDLE   hThread = 0;
DWORD ThreadId;
hThread = CreateThread(NULL, 0, ThreadProc, 0, 0, &ThreadId); // 创建线程
```

（2）实现线程工作函数 ThreadProc，代码如下：

```
DWORD WINAPI ThreadProc(void *pParam)
{
    HRESULT hr;
    HANDLE  EventHandles[2];
    HANDLE  hEvent = NULL;
    DWORD   WaitStatus;
    LONG    State;
    // 第 1 次调用 GetStatus，得到操作指令完成指示的事件句柄（当一个操作指令完成后，
    // 这个事件状态变为有效）
    hr = MyDevCap.pTransport->GetStatus(ED_NOTIFY_HEVENT_GET, (long *)
&hEvent);
    while (hThread && hEvent && hThreadEndEvent) {
        EnterCriticalSection(&csIssueCmd);
            State = 0;
            // 第 2 次调用 GetStatus，试图获取当前 DV 机状态
            hr = MyDevCap.pTransport->GetStatus(ED_MODE_CHANGE_NOTIFY,
&State);
```

```
            LeaveCriticalSection(&csIssueCmd);

            if(hr == E_PENDING) {       // 返回E_PENDING说明当前DV机支持事件通知
                EventHandles[0] = hEvent;
                EventHandles[1] = hThreadEndEvent;
                WaitStatus = WaitForMultipleObjects(2, EventHandles, FALSE,
INFINITE);
                if(WAIT_OBJECT_0 == WaitStatus) {
                    // 这是一个自定义函数，根据当前状态刷新应用程序界面
                    UpdateTransportState(State);
                }
                else {
                    break;  // End this thread
                }
            }
            else {
                break;  // End this thread (Device does not support
    notification.)
            }
        } // while
        // 取消事件通知操作
        hr = MyDevCap.pTransport->GetStatus(ED_NOTIFY_HEVENT_RELEASE, (long
*) &hEvent);
        return 1;
    }
```

（3）当 DV 机不支持事件通知时，即第 2 次调用 IAMExtTransport::GetStatus 返回值不是 E_PENDING，我们需要再使用另外的线程（或者定义一个窗口定时器）去定时检查 DV 机状态。这个线程工作函数实现代码如下：

```
    DWORD WINAPI ThreadProc(void *pParam)
    {
        HRESULT hr;
        LONG State;
        DWORD WaitStatus;
        while (hThread && hThreadEndEvent)
        {
            EnterCriticalSection(&csIssueCmd);
                State = 0;
                // 调用get_Mode获取当前DV机状态
                hr = MyDevCap.pTransport->get_Mode(&State);
            LeaveCriticalSection(&csIssueCmd);
            // 这是一个自定义函数，根据当前状态刷新应用程序界面
            UpdateTransportState(State);

            // 等待200ms，并检查线程退出事件，准备下一次的DV状态检查
            WaitStatus = WaitForSingleObjectEx(hThreadEndEvent, 200, FALSE);
            if (WaitStatus == WAIT_OBJECT_0)
                break; // Exit thread now
                    // Otherwise, the wait timed out. Time to poll again
```

```
    }
    return 1;
}
```

（4）在应用程序退出前，终止上述辅助线程，代码如下：

```
if (hThread)
{
    // Signaling this event will cause the thread to end.
    if (SetEvent(ThreadEndEvent)) {
        // Wait for it to end.
        WaitForSingleObjectEx(hThread, INFINITE, FALSE);
    }
}
CloseHandle(hThreadEndEvent);
CloseHandle(hThread);
```

提示 在上述过程中，当我们发出一个新的 DV 状态设置时，也应该使用 csIssueCmd 同步对象。代码如下：

```
// Issue the "stop" command
EnterCriticalSection(&csIssueCmd);
    if(hr = MyDevCap.pTransport->put_Mode(ED_MODE_STOP), SUCCEEDED(hr))
        UpdateTransportState(ED_MODE_STOP);
LeaveCriticalSection(&csIssueCmd);
```

最后，我们来看一下磁带播放时的时间戳（Time Code）问题。我们可以通过接口方法 IAMTimecodeReader::GetTimecode 来获得，它使用了一个数据结构来描述时间戳，代码如下：

```
typedef struct tagTIMECODE_SAMPLE {
    LONGLONG   qwTick;
    TIMECODE   timecode;
    DWORD      dwUser;
    DWORD      dwFlags;
} TIMECODE_SAMPLE;

typedef struct tagTIMECODE {
    WORD    wFrameRate;
    WORD    wFrameFract;
    DWORD   dwFrames;
} TIMECODE;
```

值得注意的是，时间戳以一个 DWORD 类型的数据表示，时间的各个部分（小时、分、秒、帧）都是 BCD 码，并按 *hhmmssff* 顺序排列。代码如下：

```
if (MyDevCap.bHasTimecode)
{
    TIMECODE_SAMPLE TimecodeSample;
    TimecodeSample.timecode.dwFrames = 0;
    char szBuf[32];
    TimecodeSample.dwFlags = ED_DEVCAP_TIMECODE_READ;     // 初始化
```

```
    if (hr = MyDevCap.pTimecode->GetTimecode(&TimecodeSample),
SUCCEEDED(hr))
    {
        DWORD dwTime = TimecodeSample.timecode.dwFrames; // BCD 码格式
        int hour  = ((dwTime & 0x0F000000) >> 24) +
                    (10 * ((dwTime & 0xF0000000) >> 28));
        int min   = ((dwTime & 0x0F0000) >> 16) +
                    (10 * ((dwTime & 0xF00000) >> 20));
        int sec   = ((dwTime & 0x0F00) >> 8) +
                    (10 * ((dwTime & 0xF000) >> 12));
        int frame = (dwTime & 0x0F) +
                    (10 * ((dwTime & 0xF0) >> 4));
    }
    TimecodeSample.dwFlags = ED_DEVCAP_ATN_READ; // 初始化，读取当前轨道索引
    if (hr = MyDevCap.pTimecode->GetTimecode(&TimecodeSample),
SUCCEEDED(hr))
    {
        DWORD dwTrackNumber = TimecodeSample.timecode.dwFrames;
    }
}
```

8.1.2　磁带的录像

如果我们想将本地硬盘上的 DV 文件内容放到磁带上去，应该怎么操作呢？

我们首先将 DV 机打开，并置于 VTR 模式（即磁带播放模式），然后按如下步骤操作：

（1）创建一个 DV 机 Filter;

（2）初始化 Capture Graph Builder 组件；

（3）根据 DV 文件类型（Type-1 或 Type-2），创建不同的 Filter Graph;

（4）将 DV 机置于（录像）暂停状态；

（5）将 Filter Graph 置于暂停状态；

（6）将 DV 机置于录像状态，然后运行 Filter Graph。

提示　DV 机从暂停状态转到录像状态是需要一定时间的。上述操作顺序可能会导致视频内容录到磁带上后，开始部分有重复帧现象，但这总比丢帧好！

如图 8.9 所示为将 Type-1 类型的 DV 数据传送到磁带上应用的 Filter Graph 参考图。

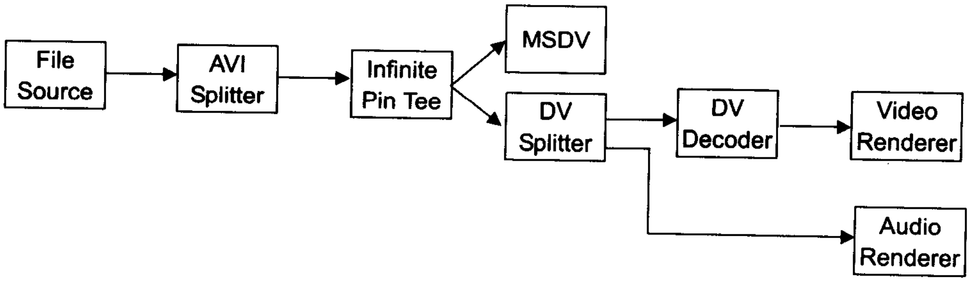

图 8.9　Type-1 类型的 DV 数据传送到磁带上

构建这种 Filter Graph 的简单代码如下：

```
ICaptureGraphBuilder2 *pBuilder;  // Capture graph builder
IBaseFilter           *pDV;       // DV capture filter (MSDV)
// Initialize pDV (not shown)
// Create and initialize the Capture Graph Builder (not shown)
// Add the Infinite Pin Tee filter to the graph
IBaseFilter *pTee;
hr = CoCreateInstance(CLSID_InfTee, 0, CLSCTX_INPROC_SERVER
    IID_IBaseFilter, reinterpret_cast<void**>(&pTee));
hr = pGraph->AddFilter(pTee, L"Tee");
// Add the File Source filter to the graph
IBaseFilter *pFileSource;
hr = pGraph->AddSourceFilter(
    L"C:\\YourFileNameHere.avi",
    L"Source",
    &pFileSource);
// Add the AVI Splitter filter to the graph
IBaseFilter *pAviSplit;
hr = CoCreateInstance(CLSID_AviSplitter, 0, CLSCTX_INPROC_SERVER
    IID_IBaseFilter, reinterpret_cast<void**>(&pAviSplit));
hr = pGraph->AddFilter(pAviSplit, L"AVI Splitter");
// Connect the file source to the AVI Splitter
hr = pBuilder->RenderStream(0, 0, pFileSource, 0, pAviSplit);
if (FAILED(hr))
{
    // This is not an AVI file
}
// Connect the file source to the Infinite Pin Tee
hr = pBuilder->RenderStream(0, &MEDIATYPE_Interleaved, pAviSplit, 0,
pTee);
if (FAILED(hr))
{
    // This is not a type-1 DV file
}
// Render one stream from the Infinite Pin Tee to MSDV, for transmit
hr = pBuilder->RenderStream(0, 0, pTee, 0, pDV);
// Render another stream from the Infinite Pin Tee for preview
hr = pBuilder->RenderStream(0, 0, pTee, 0, 0);
```

如图 8.10 所示为将 Type-2 类型的 DV 数据传送到磁带上应用的 Filter Graph 参考图。

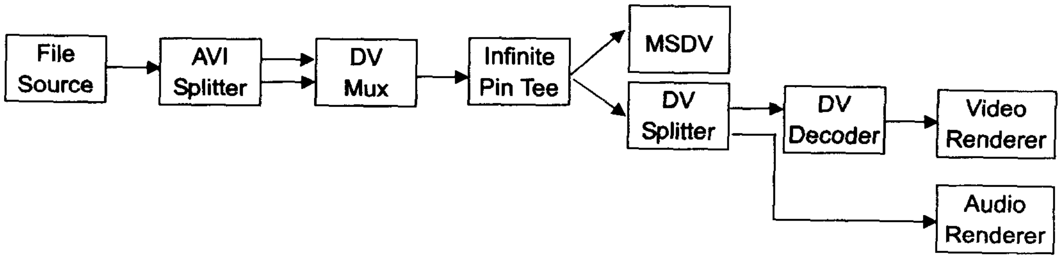

图 8.10 Type-2 类型的 DV 数据传送到磁带上

构建这种 Filter Graph 的简单代码如下：

```
// Add the DV Mux filter to the graph
IBaseFilter *pDVMux;
hr = CoCreateInstance(CLSID_DVMux, 0, CLSCTX_INPROC_SERVER
    IID_IBaseFilter, reinterpret_cast<void**>(&pDVMux));
hr = pGraph->AddFilter(pDVMux, L"DV Mux");
// Add the File Source filter to the graph
IBaseFilter *pFileSource;
hr = pGraph->AddSourceFilter(L"C:\\Example2.avi", L"Source",
    &pFileSource);
hr = pBuilder->RenderStream(0, 0, pFileSource, 0, pDVMux);
hr = pBuilder->RenderStream(0, 0, pFileSource, 0, pDVMux);
// Add the Infinite Pin Tee filter to the graph
IBaseFilter *pTee;
hr = CoCreateInstance(CLSID_InfTee, 0, CLSCTX_INPROC_SERVER
    IID_IBaseFilter, reinterpret_cast<void**>(&pTee));
hr = pGraph->AddFilter(pTee, L"Tee");

hr = pBuilder->RenderStream(0, 0, pDVMux, 0, pTee);
hr = pBuilder->RenderStream(0, 0, pTee, 0, pDV);
hr = pBuilder->RenderStream(0, &MEDIATYPE_Interleaved, pTee, 0, 0);
```

8.1.3　DV 的采集

DV 采集是指将 DV 机 Filter 输出的数据保存到本地文件中。视频来源可能是摄像头，也可能是磁带。

创建 DV 采集应用的一般步骤如下：

（1）创建一个 DV 机 Filter；

（2）初始化 Capture Graph Builder 组件；

（3）根据不同的应用（比如采集生成的文件类型是 Type-1、Type-2 或是非压缩格式），构建不同的 Filter Graph；

（4）运行 Filter Graph。

提示　与摄像头实时采集模式相比，从磁带上抓取内容除了多了对磁带播放的控制外，其他操作类似。另外，当从磁带上抓取内容时，为了避免丢帧，应该先运行 Filter Graph，然后播放磁带；而当结束采集时，应该先停止磁带播放，然后停止 Filter Graph。

如图 8.11 所示为采集生成Type-1 类型的文件格式时应用的 Filter Graph 参考图。

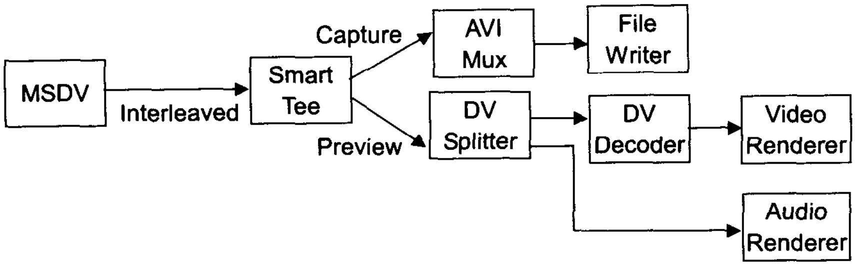

图 8.11　采集生成 Type-1 类型的 AVI 文件

构建这种 Filter Graph 的简单代码如下：

```
ICaptureGraphBuilder2 *pBuilder;   // Capture graph builder
IBaseFilter           *pDV;        // DV capture filter (MSDV)
IBaseFilter           *pAviMux     // Avi Mux filter
// Initialize pDV (not shown)
// Create and initialize the Capture Graph Builder (not shown)
// Create the file-writing section of the graph
hr = pBuilder->SetOutputFileName(&MEDIASUBTYPE_Avi,
    OLESTR("C:\\Example1.avi"), &pAviMux, 0);
// Render the capture stream
hr = pBuilder->RenderStream(&PIN_CATEGORY_CAPTURE,
&MEDIATYPE_Interleaved,
    pDV, 0, pAviMux);
// Render the preview stream
hr = pBuilder->RenderStream(&PIN_CATEGORY_PREVIEW,
&MEDIATYPE_Interleaved,
    pDV, 0, 0);
// Remember to release all interfaces
```

如图 8.12 所示为采集生成 Type-2 类型的文件时应用的 Filter Graph 参考图。

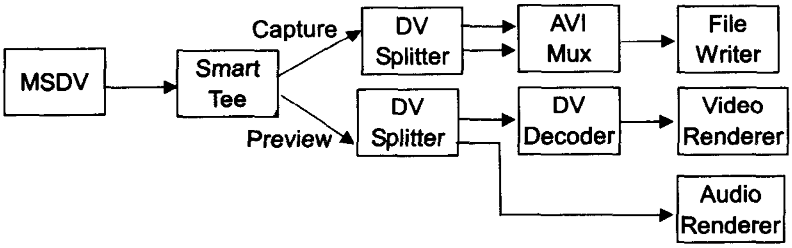

图 8.12　采集生成 Type-2 类型的 AVI 文件

构建这种 Filter Graph 的简单代码如下：

```
ICaptureGraphBuilder2 *pBuilder;   // Capture graph builder
IBaseFilter           *pDV;        // DV capture filter (MSDV)
IBaseFilter           *pAviMux     // Avi Mux filter
IBaseFilter           *pDVSplit;   // DV Splitter

// Initialize pDV (not shown)
// Create and initialize the Capture Graph Builder (not shown)
```

```
// Create the DV Splitter and add it to the filter graph.
hr = CoCreateInstance(CLSID_DVSplitter, 0, CLSCTX_INPROC_SERVER,
    IID_IBaseFilter, reinterpret_cast<void**>(&pDVSplit));
hr = pGraph->AddFilter(pDVSplit, L"DV Splitter");
// Create the file-writing section of the graph
hr = pBuilder->SetOutputFileName(&MEDIASUBTYPE_Avi,
    OLESTR("C:\\Example2.avi"), &pAviMux, 0);
// Connect the capture pin to the DV Splitter, and render one stream
// from the DV Splitter to the AVI Mux
hr = pBuilder->RenderStream(&PIN_CATEGORY_CAPTURE,
&MEDIATYPE_Interleaved, pDV, pDVSplit, pAviMux);
// Render the other stream from the DV splitter to the AVI Mux
hr = pBuilder->RenderStream(0, 0, pDVSplit, 0, pAviMux);
// Render the preview stream
hr = pBuilder->RenderStream(&PIN_CATEGORY_PREVIEW,
&MEDIATYPE_Interleaved,
    pDV, 0, 0);

// Remember to release all interfaces
```

如图 8.13 所示为采集生成非压缩格式的 AVI 文件应用的 Filter Graph 参考图。

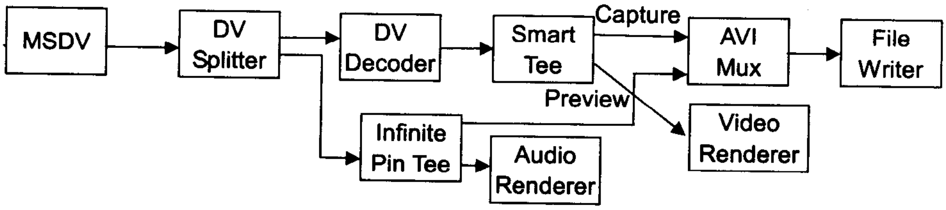

图 8.13　采集生成非压缩格式的 AVI 文件

构建这种 Filter Graph 的简单代码如下：

```
// Build the file-writing section of the graph
hr = pBuilder->SetOutputFileName(&MEDIASUBTYPE_Avi,
    OLESTR("C:\\Example3.avi"), &pMux, 0);
// MSDV to DV splitter
IBaseFilter *pDVSplit;  // Create the DV Splitter (CLSID_DVSplitter)
hr = pBuilder->RenderStream(0, &MEDIATYPE_Interleaved, pDV, 0, pDVSplit);
// Splitter to DV Decoder to Smart Tee
IBaseFilter *pDVDec; // Create the DV Decoder (CLSID_DVVideoCodec)
IBaseFilter *pSmartTee; // Create the Smart Tee (CLSID_SmartTee)
hr = pBuilder->RenderStream(0, &MEDIATYPE_Video, pDVSplit, pDVDec,
    pSmartTee);
// Smart Tee (video) to Avi Mux
IPin *pPin1;
hr = pBuilder->FindPin(pSmartTee, PINDIR_OUTPUT, 0, 0, TRUE, 0, &pPin1);
hr = pBuilder->RenderStream(0, 0, pPin1, 0, pMux);
// Smart Tee to preview
IPin *pPin2;
hr = pBuilder->FindPin(pSmartTee, PINDIR_OUTPUT, 0, 0, TRUE, 1, &pPin2);
```

```
hr = pBuilder->RenderStream(0, 0, pPin2, 0, pMux);
// DV Splitter (audio) to Infinite Tee to Avi Mux
IBaseFilter *pTee; // Create the Infinite Pin Tee (CLSID_InfTee)
hr = pBuilder->RenderStream(0, &MEDIATYPE_Audio, pDVSplit, pTee, pMux);

// Infinite Pin Tee to preview
hr = pBuilder->RenderStream(0, 0, pTee, 0, 0);
```

8.2 实例解剖

DirectShow SDK 提供了一个 DV 机应用的例子，这个例子位于 SDK 安装目录的 Samples\C++\DirectShow\Capture\DVApp 下。

8.2.1 实现的功能

如图 8.14 所示是 DVApp 例子的程序界面。这个应用程序完成的主要功能有：

- 控制 DV 摄像机进行磁带播放，摄像头预览等；
- 采集 DV 数据，保存为 Type-1 或 Type-2 类型的 AVI 文件；
- 将 Type-1 或 Type-2 的 DV 文件数据转录到磁带上。

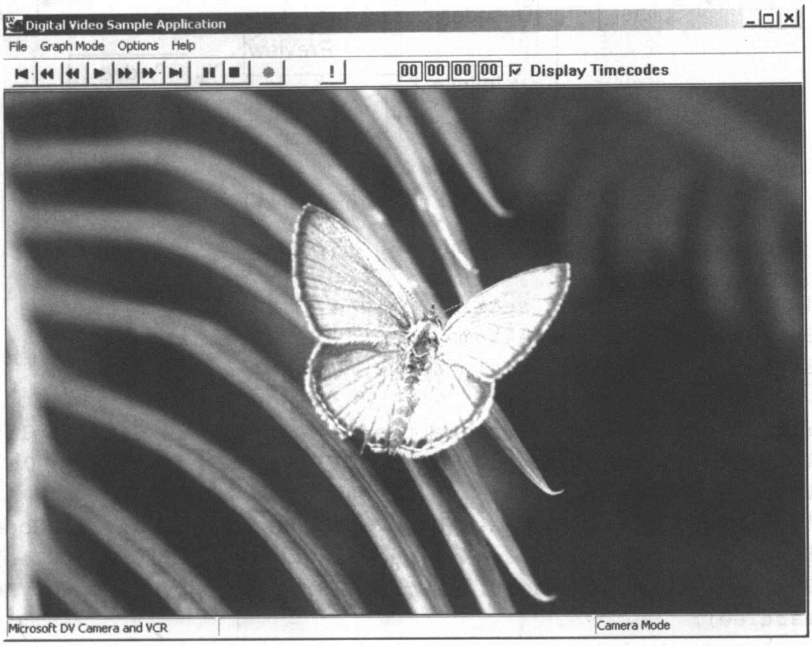

图 8.14 DVApp 例子的程序界面

8.2.2 实现要点

通过 DVApp 例子的学习，我们将掌握如下知识点：

- 创建磁带播放、磁带录像、DV 采集等不同应用的 Filter Graph；

- 刷新 DV 机的状态；
- 在磁带播放时显示时间戳；
- 改变 DV 解码后输出的视频图像大小；
- 改变 DV 采集输出的帧率；
- 对 DV 机直接发送指令；
- 保存应用程序中创建的 Filter Graph。

创建磁带播放、磁带录像、DV 采集等不同应用的 Filter Graph

DVApp 例子是一个标准的 Win32 应用程序。程序执行的流程是：WinMain | FInitMain | DV_AppSetup | NMsgLoop。其中，FInitMain 完成了应用程序主窗口的初始化；NMsgLoop 完成了主线程的消息循环；DV_AppSetup 完成了 DV 相关的一些初始化，它的内部流程是 g_pGraph = new CDVGraph() | g_pGraph->BuildBasicGraph() | DV_RefreshMode() | g_pGraph->MakePreviewGraph()……

我们看到，DVApp 例子将 Filter Graph 相关的操作都封装在了一个 CDVGraph 类里面（我们可以查看 CDVGraph.h 的定义），并定义了 9 种 Filter Graph，代码如下：

```
enum GRAPH_TYPE
{
    GRAPH_PREVIEW,                    // 单纯的预览
    GRAPH_DV_TO_FILE,                 // DV 采集到 Type-1 文件（同时预览）
    GRAPH_DV_TO_FILE_NOPRE,           // DV 采集到 Type-1 文件（没有预览）
    GRAPH_FILE_TO_DV,                 // Type-1 文件传送到磁带（同时预览）
    GRAPH_FILE_TO_DV_NOPRE,           // Type-1 文件传送到磁带（没有预览）
    GRAPH_DV_TO_FILE_TYPE2,           // DV 采集到 Type-2 文件（同时预览）
    GRAPH_DV_TO_FILE_NOPRE_TYPE2,     // DV 采集到 Type-2 文件（没有预览）
    GRAPH_FILE_TO_DV_TYPE2,           // Type-2 文件传送到磁带（同时预览）
    GRAPH_FILE_TO_DV_NOPRE_TYPE2      // Type-2 文件传送到磁带（没有预览）
};
```

每一种 Filter Graph 都有对应的构建函数，代码如下：

```
HRESULT MakePreviewGraph(void);
// Type 1 File (capture\playback\transmit)
HRESULT MakeDvToFileGraph_Type1(TCHAR*        OutputFileName);
HRESULT MakeDvToFileGraph_NoPre_Type1(TCHAR*  OutputFileName);
HRESULT MakeFileToDvGraph_Type1(TCHAR*        InputFileName);
HRESULT MakeFileToDvGraph_NoPre_Type1(TCHAR*  InputFileName);
// Type 2 File (capture\playback\transmit)
HRESULT MakeDvToFileGraph_Type2(TCHAR*        OutputFileName);
HRESULT MakeDvToFileGraph_NoPre_Type2(TCHAR*       OutputFileName);
HRESULT MakeFileToDvGraph_Type2(TCHAR*  InputFileName);
HRESULT MakeFileToDvGraph_NoPre_Type2(TCHAR*  InputFileName);
```

在构建具体 Filter Graph 的时候，大部分都使用了 ICaptureGraphBuilder::RenderStream 接口方法，比如 MakePreviewGraph 函数的实现代码如下：

```
HRESULT CDVGraph::MakePreviewGraph()
{
```

```
    m_iGraphType = GRAPH_PREVIEW;
    HRESULT hr = m_pCaptureGraphBuilder->RenderStream(&PIN_CATEGORY
_PREVIEW, &MEDIATYPE_Interleaved, m_pDeviceFilter, NULL, NULL);
    return hr;
}
```

DV 机状态的刷新

DV 机状态的刷新，见 DVApp 例子 dvapp.cpp 文件的 DV_RefreshMode 函数实现。代码如下：

```
// Query the current device type
DV_MODE SubunitMode;
g_pGraph->GetDVMode( &SubunitMode );
```

这段代码取得 DV 机当前是处于 CameraMode 还是 VcrMode，并根据不同的模式刷新界面元素的状态。

另外，工具条上的各个控制按钮对应的操作响应，以及它们的状态刷新，可以参考 DV_TransportCommand_WndProc 函数的实现。

在磁带播放时显示时间戳

在 DVApp 例子中，在 DV_TransportCommand_WndProc 函数的 case IDM_PLAY 响应中，也就是用户按下了工具条上的“播放”按钮时，根据是否需要显示时间戳，决定是否启动一个定时器，用于不断读取当前的播放位置，代码如下：

```
if (IsDlgButtonChecked(g_hwndTBar, IDC_TCCHECKBOX))
{
    SetTimer(hwnd, DV_TIMER_ATN,  DV_TIMERFREQ, (TIMERPROC)
DV_TimecodeTimerProc);
    g_bUseAtnTimer = TRUE;
}
```

定时器处理函数 DV_TimecodeTimerProc 的代码如下：

```
void CALLBACK DV_TimecodeTimerProc(HWND hwnd, UINT uMsg, UINT idEvent,
DWORD dwTime)
{
    DV_DisplayTimecode();
}
void DV_DisplayTimecode(void)
{
    TIMECODE_SAMPLE TimecodeSample;
    TimecodeSample.timecode.dwFrames = 0;
    static DWORD i1 = 0, i2 = 0, i3 = 0;
    TCHAR szBuf[4];
    TimecodeSample.dwFlags = ED_DEVCAP_TIMECODE_READ;
    // Query the TimeCode sample data
    HRESULT hr = g_pGraph->m_pIAMTCReader->GetTimecode(&TimecodeSample);
    if(FAILED(hr))
    {
```

```
        // MBOX(TEXT("DV_DisplayTimecode::g_pGraph->m_pIAMTCReader->
        // GetTimecode()failed"));
    }
    // 从 DWORD 型的 BCD 码中分离出时、分、秒、帧各个部分，然后在界面上显示
    wsprintf(szBuf, TEXT("%.2x"),((TimecodeSample.timecode.dwFrames &
0xff000000) >> 24));
    SetDlgItemText(g_hwndTBar, IDC_EDIT_HOUR, szBuf);
    wsprintf(szBuf, TEXT("%.2x"),((TimecodeSample.timecode.dwFrames &
0x00ff0000) >> 16));
    SetDlgItemText(g_hwndTBar, IDC_EDIT_MINUTE, szBuf);
    wsprintf(szBuf, TEXT("%.2x"),((TimecodeSample.timecode.dwFrames &
0x0000ff00) >>  8));
    SetDlgItemText(g_hwndTBar, IDC_EDIT_SECOND, szBuf);
    wsprintf(szBuf, TEXT("%.2x"),(TimecodeSample.timecode.dwFrames &
0x000000ff));
    SetDlgItemText(g_hwndTBar, IDC_EDIT_FRAME, szBuf);
}
```

DV 解码后输出视频图像大小的改变

DV 图像的原始大小，NTSC 制式的为 720×480，PAL 制式的为 720×576。我们也可以通过 DV Video Decoder Filter 上的 IIPDVDec 接口来改变 DV 解码后输出图像的大小。IIPDVDec::put_IPDisplay 接口方法的参数取值如下：

```
enum _DVDECODERRESOLUTION {
    DVDECODERRESOLUTION_720x480     = 1000,
    DVDECODERRESOLUTION_360x240     = 1001,
    DVDECODERRESOLUTION_180x120     = 1002,
    DVDECODERRESOLUTION_88x60       = 1003
};
enum _DVRESOLUTION {
    DVRESOLUTION_FULL     = 1000,// 原始大小
    DVRESOLUTION_HALF     = 1001,// 宽高均缩小为一半（实际图像大小为原来的 1/4）
    DVRESOLUTION_QUARTER = 1002,// 宽高均缩小为 1/4（实际图像大小为原来的 1/16）
    DVRESOLUTION_DC       = 1003 // 宽高均大致缩小为 1/8（实际图像大小为原来的 1/64）
};
```

put_IPDisplay 参数的取值为上述某个枚举类型后，实际输出图像的大小取决于实际视频的制式，见表 8.2。

表 8.2　枚举类型与制式的对应关系

枚举类型值	NTSC	PAL
DVRESOLUTION_FULL	720×480	720×576
DVRESOLUTION_HALF	360×240	360×288
DVRESOLUTION_QUARTER	180×120	180×144
DVRESOLUTION_DC	88×60	88×72

在 DVApp 例子中，这部分的实现参见 CDVGraph::GetVideoWindowDimensions 函数和 CDVGraph::GetResolutionFromDVDecoderPropertyPage 函数。不过，并没有调用接口方法 IIPDVDec::put_IPDisplay，而是将 DV Video Decoder 的属性页显示出来，让用户直接进行设置。

DV 采集输出帧率的改变

在一些实时处理的场合，由于处理器的压力过大，我们可能需要降低 DV 视频流输出的帧率。我们可以通过 DV Splitter 上的 IDVSplitter 接口来实现这一点。比如 NTSC 制式的 DV，原来是 30fps，我们可以让它只输出 15fps；PAL 制式的，原来是 25fps，我们可以让它只输出 12.5fps。

DVApp 例子的具体实现见 CDVGraph::ChangeFrameRate，代码如下：

```
HRESULT CDVGraph::ChangeFrameRate(BOOL bHalfFrameRate)
{
    HRESULT hr = S_OK;
    CComPtr<IBaseFilter>    pDVSplitter;
    CComPtr<IDVSplitter>    pIDVSplitter = NULL;
    /* Obtain the dv docoder's IBaseFilter interface. */
    hr = m_pGraph->FindFilterByName(L"DV Splitter", &pDVSplitter) ;
    CHECK_ERROR( TEXT("CDVGraph::ChangeFrameRate()::m_pGraph->
FindFilterByName failed."), hr);
    hr = pDVSplitter->QueryInterface(IID_IDVSplitter,
reinterpret_cast<PVOID *>(&pIDVSplitter));
    CHECK_ERROR( TEXT("CDVGraph::ChangeFrameRate()::QI IDVSplitter
failed."), hr);
    if(bHalfFrameRate)
        hr = pIDVSplitter->DiscardAlternateVideoFrames(1);
// if the value is non-zero, discards alternate frames
    else
        hr = pIDVSplitter->DiscardAlternateVideoFrames(0);
// If the value is zero, the filter delivers every frame
    CHECK_ERROR( TEXT("CDVGraph::ChangeFrameRate()::pIDVSplitter->
DiscardAlternateVideoFrames failed."), hr);
    return hr;
}
```

对 DV 机直接发送指令

我们已经知道，IAMExtDevice、IAMExtTransport 和 IAMTimecodeReader 的大部分接口方法的实现，都是在 DV 机 Filter 内部转化成相应的指令发送给驱动程序的。另外，我们还有一种直接发送指令的方法，就是调用 IAMExtTransport::GetTransportBasicParameters。我们可以参见 CDVGraph::SeekATN 函数的实现。

保存应用程序中创建的 Filter Graph

为了调试方便，我们有时候需要将应用程序创建的 Filter Graph 保存为 GraphEdit 能够打开的格式文件（.grf）。在 DVApp 例子中，我们可以参见 CDVGraph::SaveGraphToFile 函数的实现，代码如下：

```
HRESULT CDVGraph::SaveGraphToFile(TCHAR* sGraphFile)
{
    IStorage *          pStorage = NULL;
    IStream *           pStream = NULL;
```

```
    IPersistStream *    pPersistStream = NULL;
    HRESULT             hr = S_OK;
    if(m_pCaptureGraphBuilder == NULL || sGraphFile == NULL)
        return E_FAIL;
    USES_CONVERSION;
    // Either Open or Create the *.GRF file
    hr = StgOpenStorage( T2W (sGraphFile), NULL, STGM_TRANSACTED |
STGM_READWRITE | STGM_SHARE_DENY_WRITE,
                        NULL, NULL, &pStorage );
    if ( STG_E_FILENOTFOUND == hr )
        hr = StgCreateDocfile( T2W (sGraphFile), STGM_CREATE |
STGM_TRANSACTED | STGM_READWRITE | STGM_SHARE_EXCLUSIVE ,
                        NULL , &pStorage);
    CHECK_ERROR( TEXT(" StgCreateDocfile failed."), hr);
    hr = pStorage->CreateStream( L"ActiveMovieGraph", STGM_WRITE |
STGM_CREATE | STGM_SHARE_EXCLUSIVE,
                                NULL, NULL, &pStream );
    CHECK_ERROR( TEXT(" CreateStream failed."), hr);
    // 取得 Filter Graph Manager 上的 IpersistStream 接口
    hr = m_pGraph->QueryInterface( IID_IPersistStream, (void **)
&pPersistStream );
    CHECK_ERROR( TEXT(" QI IPersistStream failed."), hr);
    // 保存当前的 Filter Graph
    hr = pPersistStream->Save(pStream, TRUE);
    CHECK_ERROR( TEXT(" pPersistStream->Save() failed."), hr);
    hr = pStorage->Commit( STGC_DEFAULT );
    CHECK_ERROR( TEXT(" Save GRF file failed."), hr);
    SAFE_RELEASE(pStorage);
    SAFE_RELEASE(pStream);
    SAFE_RELEASE(pPersistStream);
    return hr;
}
```

第 9 章　非线性编辑 DES

9.1　DES 概 述

DES（DirectShow Editing Services）是一套基于 DirectShow 核心框架的编程接口。它的出现简化了视频编辑任务，弥补了 DirectShow 对媒体文件非线性编辑支持的先天性不足。不过，就技术本身而言，DES 并没有超越 DirectShow Filter 架构，只是它之上的一层增强应用。下面，我们将要详细介绍 DES 编程中的各种新概念。

9.1.1　时间线模型

DES 的系统结构，如图 9.1 所示。

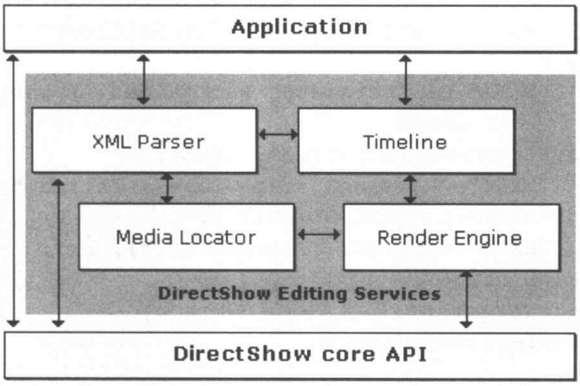

图 9.1　DES 的系统结构

- Timeline：组织各个媒体源、音视频效果、过渡处理等的信息集合，实际代表了最终的视频编辑作品。
- XML Parser：将 Timeline 结构转化为 XML 格式的文件保存，或者从 XML 格式文件生成对应的 Timeline。
- Render Engine：将 Timeline 实际转化为 DirectShow Filter 实现的输出控制引擎。
- Media Locator：用于媒体文件的定位。

我们看到，上述 4 个主要部分组成了 DES，而 DES 介于 Application 和 DirectShow core API 之间，是 DirectShow 上的一层增强应用。

下面，我们举个例子来看一下 DES 能够给我们带来些什么。假如我们现在有 3 个媒体文件 A、B 和 C，现在要使用这 3 个文件合成一个新的文件。我们想取 A 的 4s 的内容，紧接着取 B 的 10s 的内容，再紧接着 C 的 5s 的内容。如果仅仅是这样，我们直接使用 DirectShow Filter 是不难实现的。

提示 一种可行的方法是，我们在应用程序上维持各个文件的编辑信息，由应用程序根据这些信息动态创建、控制功能单一的 Filter Graph，并按顺序对各个文件进行处理。

但事实上，“创意”是随时会改变的，假设我们现在想让 C 在 B 之前出现，或者我们想取 A 的不同位置的 10s 内容，或者我们想给整个合成的文件加上一段美妙的背景音乐。如果我们仍然直接使用 DirectShow Filter 去实现，情况就变得很复杂了。然而对于 DES 来说，这真的是“小菜一碟”，即将所有的编辑信息以 DES 提供的接口“下放”。就这么简单！（其他的如 Filter Graph 的创建、控制输出，就完全交给 DES 来处理吧！）

使用 DES 的好处还有很多，简单罗列如下：

（1）基于时间线（Timeline）的结构以及轨道（Track）的概念，使得多媒体文件的组织、编辑变得直观而高效；

（2）支持即时的预览；

（3）视频编辑项目支持以 XML 格式文档的保存；

（4）支持对视频、音频的效果处理，以及视频之间切换的过渡处理；

（5）可以直接使用 DES 提供的 100 多种 SMPTE 过渡（Transition）处理，以及微软公司的 IE 自带的各种音视频效果（Effect）、过渡处理组件；

（6）支持通过色调、亮度、RGB 值或者 Alpha 值进行图像的合成；

（7）自动对源文件输出的视频帧率、音频采样率进行调整，直接支持视频的缩放。

DES 的内部结构模型（一个基于时间线的模型）如图 9.2 所示。

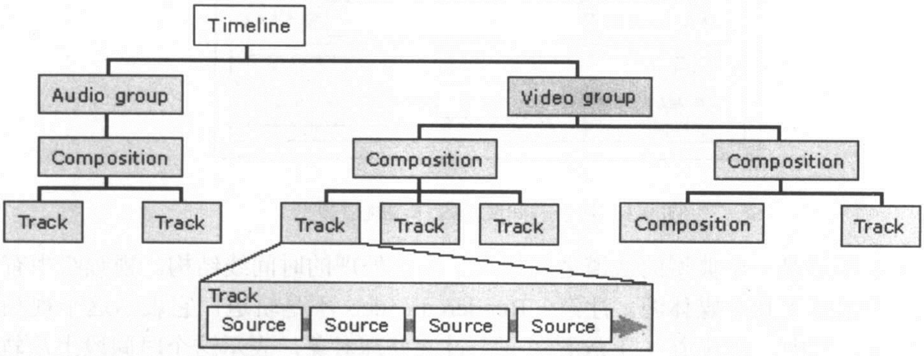

图 9.2　DES 的内部结构模型

我们看到，这是一个树形结构。（大部分出色的管理都使用树形结构。）在这棵树中，音、视频文件是叶节点，称为（媒体）源（图中标识为 Source）；一个或多个源放在一个轨道（图中标识为 Track）中，每个轨道都有统一的媒体格式输出；轨道的集合称为复合节点（图中标识为 Composition），每个复合节点都可以对其所有的子复合节点或轨道进行各种复杂的编辑；顶级的复合节点或轨道构成了组（图中标识为 Group）；每个组输出单一格式的媒体流，所有的组构成一个时间线（图中标识为 Timeline）。时间线表示一个视频编辑的项目，它是这棵树的根节点。一个时间线项目必须至少包含一个组，最典

型的情况下会有两个组：音频组（Audio Group）和视频组（Video Group）。

提示

在 DES 中，音频和视频是分开处理的。因此，一个同时包含有音频和视频的媒体文件，如果要在 DES 中编辑后重现，我们需要将这个媒体文件各加入音频组和视频组一次（加入音频组的仅输出音频流，加入视频组的仅输出视频流）。

如图 9.3 所示是一个典型的基于时间线的媒体源轨道编排，箭头方向即是时间流逝的方向。这个时间线由两个组构成，每个组中包含两条轨道。在组中，轨道是有优先级的（轨道 0 具有最低的优先级，轨道 1 稍高一点，依次类推）。运行时，总是先输出高优先级的轨道中的媒体源内容。如果此时高优先级的轨道中没有媒体源的内容输出，则让低优先级的轨道中的媒体源输出。图中视频组的输出顺序为 Source A→Source C→Source B；而对于音频组，它的所有轨道的输出都只是简单的合成。

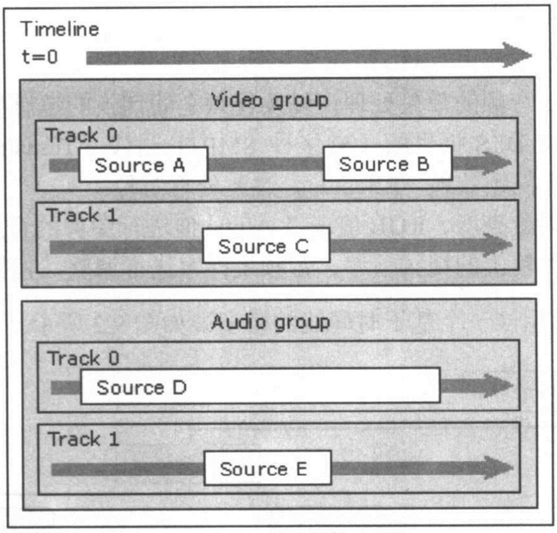

图 9.3　基于时间线的媒体源轨道模型

如图 9.4 所示是一个典型的轨道之间加入了过渡处理的时间线结构。视频组中有两个轨道，轨道上编排了几个媒体源。注意：Rendered video 不是轨道，它表示这个视频组最终输出的效果。其中，在轨道 1 上放置了一个过渡处理对象，表示这个时间段上从轨道 0 过渡到轨道 1 的混合效果。一般过渡处理对象位于高优先级的轨道上，而且过渡也是有方向的，默认是从低优先级的轨道过渡到高优先级的轨道。当然，我们也可以改变过渡的方向。如图 9.5 所示，第 1 个过渡是从轨道 0 到轨道 1，第 2 个过渡是从轨道 1 到轨道 0。

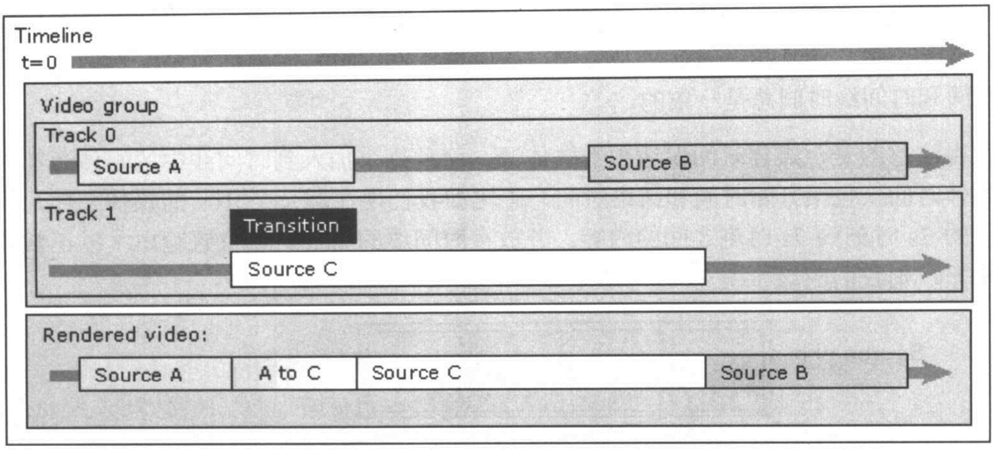

图 9.4 加入过渡处理的轨道模型

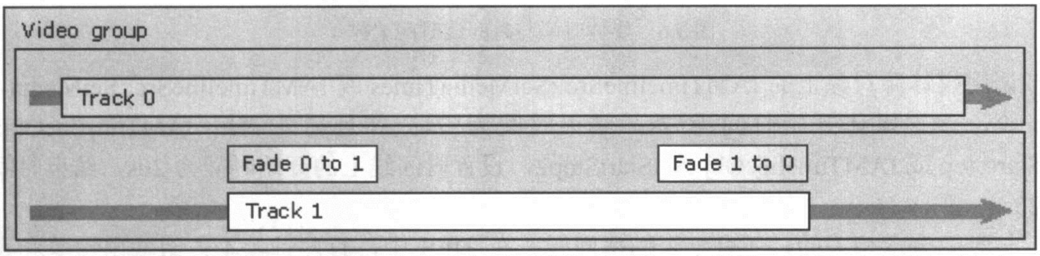

图 9.5 过渡的两种方向

从图 9.2 的时间线树形结构中不难发现，每个组下面都有一个复合节点，而这几个图例中，我们看到的组都是直接由轨道构成的。那么，复合节点到底有什么用呢？熟悉设计模式的读者很容易就明白了，DES 采用的就是“对象结构型模式”的 Composite（组合）的模式。复合节点可以包装几个轨道（这几个轨道之间可能是包含过渡处理的），组成一个虚拟轨道（Virtual Track），并且与其他普通的轨道接口保持一致。我们完全可以把这个虚拟轨道与普通的轨道一样操作，进而很方便地进行更加复杂、丰富的效果编辑。

既然最终的时间线对象是靠时间来组织起各个媒体源的，我们接下去就来介绍一下 DES 中时间的概念。

9.1.2 时间概念

DES 中大致有 3 种时间：

- 时间线时间（Timeline Time）：相对于整个时间线项目的时间，我们也可以称它为项目时间；
- 媒体时间（Media Time）：相对于媒体源的时间，比如媒体源是一个文件，那么媒体时间实际上是指相对于文件开头的位置时间；
- 父对象时间（Parent Time）：相对于时间线中已经编排的某个对象的时间，比如时间线上有一个对象在 8s 时刻开始，这个对象还包含一个实际在时间线上 10s 时

刻开始的子对象，那么这个子对象相对于父对象的开始时刻为 2s。虚拟轨道包含在时间线对象中，而且相对时间总是为 0，所以编排在虚拟轨道中的对象，它们的父对象时间和时间线时间总是一致的。

值得注意的是，媒体时间仅适用于媒体源，而且每个加入到时间线中的媒体源都必须设置媒体时间（包括开始时间和结束时间）。比如我们现在有一个 10s 的媒体文件，我们想取它的 2s 时刻到 7s 时刻之间的内容，并放在时间线的 20s 时刻位置输出（以正常的播放速率），则我们应该如图 9.6 所示设置。

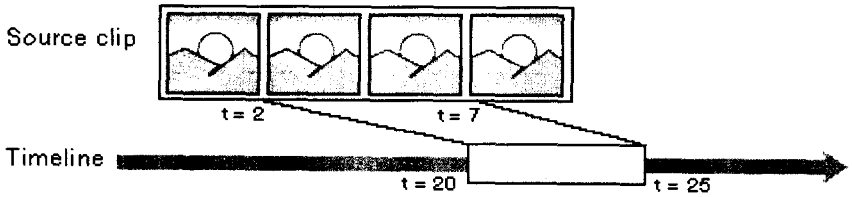

图 9.6　媒体时间与时间线时间编排

调用媒体源对象上的 IAMTimelineSrc::SetMediaTimes 或 IAMTimelineSrc::SetMediaTimes2，设置媒体开始时间为 2s，结束时间为 7s；并且另外调用 IAMTimelineObj::SetStartStop 或 IAMTimelineObj::SetStartStop2，设置时间线上的开始时间为 20s，结束时间为 25s。

与时间概念相关的，还有一个帧率问题。在 DES 中，有两种帧率：组输出帧率和媒体源帧率。前者是指在组上设置的帧率，即这个组每秒钟输出的帧数（时间线中的每个组都可以看成是一个拥有自己帧率的独立媒体流）；而后者是媒体源本身的帧率，一般不能更改。

在图 9.6 的编排中，我们在媒体源上设置的媒体时间长度与时间线时间长度是相等的，所以媒体源播放正常；如果上述两个时间长度不相等，则会引入一个“不正常”播放速率问题。

如图 9.7 所示，实际是 10s 的媒体内容，在时间线上设置的时间长度仅为 5s，那么最终输出效果是，这 10s 的视频内容以两倍速（快进）播放。

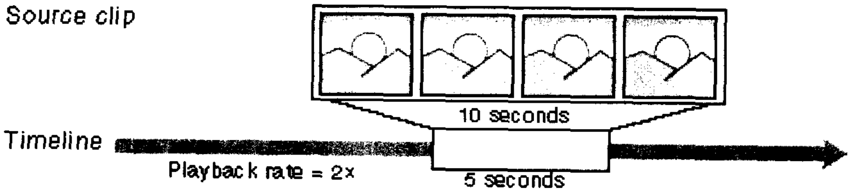

图 9.7　DES 中的播放速率问题

以下几点我们需要注意：

- 播放速率的计算公式为播放速率 = （媒体结束时间 – 媒体开始时间） / （时间线结束时间 – 时间线开始时间）；

- 我们可以通过改变媒体源上的媒体时间长度或者时间线时间长度，来达到改变播放速率的目的，这样做并不会影响到组的最终输出帧率；
- 在组上改变输出帧率，也不会影响到媒体源的播放速率；
- 在同一个组中，我们可以加入不同帧率的媒体源。DES 会对它们自动调整，以符合组上设置的输出帧率。（其实，DES 内部使用了一个实现帧率转换的 Frame Rate Converter Filter。）

最后，我们来了解一下 DES 中的帧边界处理问题。当我们输出一个时间线项目时，DES 会自动将所有的时间就近取舍为帧边界（帧边界的时间点主要取决于组上设置的输出帧率）。假如我们有一个视频组，设置的输出帧率为 30fps（它的单帧时间近似为 33 毫秒），现在我们有一个视频源文件，要取它 1.68s 长度的内容加到时间线的 0 到 1.68s 上。我们发现，1.68s 并不是一个帧边界（第 50 帧是 1.6666s，第 51 帧是 1.6999s）。于是，DES 会自动将媒体源的结束时间“认定为”1.6666s。如果我们此时在组上输出的媒体流进行随机定位到 1.68s，那么将超出媒体源的时间范围（可能会引起异常）。

提示 上面我们用了“认定为”3 个字，主要是因为 DES 进行自动的帧边界处理后，媒体源上的开始、结束时间并不会改变，即调用 IAMTimelineObj::GetStartStop 得到的时间还是当初设置进去的时间值。DES 的考虑是，因为组上的输出帧率随时都可能被改变，媒体源在时间线上的位置也可能被改变，因此，DES 需要根据原始设置的开始、结束时间再次进行帧边界处理。

DES 的自动帧边界处理会引起精确定位问题。我们推荐的解决方法是，首先调用媒体源上的 IAMTimelineObj::GetStartStop 获得原始设定的开始、结束时间，然后调用 IAMTimelineObj::FixTimes 将这两个时间转换成帧边界时间，以修正后的帧边界时间进行定位就行了！

9.1.3 媒体源

在一个视频编辑作品中，媒体源提供的是最基本的素材，其他的如轨道、复合节点、组只是提供组织结构，而音、视频效果和过渡处理也只是提供素材上的加工。在 DES 中，媒体源一般代表一个媒体文件。

在开发 DES 应用时，我们一般都要首先创建一个时间线对象，代码如下：

```
IAMTimeline *pTL = NULL;
hr = CoCreateInstance(CLSID_AMTimeline, NULL, CLSCTX_INPROC_SERVER,
        IID_IAMTimeline, (void**)&pTL);
```

然后就是使用接口方法 IAMTimeline::CreateEmptyNode 创建各种 DES 对象。代表各种 DES 对象的枚举类型如下：

```
typedef enum {
    TIMELINE_MAJOR_TYPE_COMPOSITE = 1,    // 复合节点
    TIMELINE_MAJOR_TYPE_TRACK = 2,        // 轨道
    TIMELINE_MAJOR_TYPE_SOURCE = 4,       // 媒体源
```

```
    TIMELINE_MAJOR_TYPE_TRANSITION = 8,   // 过渡
    TIMELINE_MAJOR_TYPE_EFFECT = 16,      // 音、视频效果
    TIMELINE_MAJOR_TYPE_GROUP = 128       // 组
} TIMELINE_MAJOR_TYPE;
```

调用 IAMTimeline::CreateEmptyNode 成功后，我们可以得到一个 IAMTimelineObj 接口指针。也就是说，每个 DES 对象上都实现了 IAMTimelineObj 接口，另外各个具体的对象还实现了各自特殊的接口，参考如下：

媒体源实现了 IAMTimelineSrc、IAMTimelineEffectable、IAMTimelineSplittable；

轨道实现了 IAMTimelineTrack、IAMTimelineVirtualTrack、IAMTimelineEffectable、IAMTimelineTransable、IAMTimelineSplittable；

复合节点实现了 IAMTimelineComp、IAMTimelineVirtualTrack、IAMTimelineEffectable、IAMTimelineTransable；

组实现了 IAMTimelineComp、IAMTimelineGroup；

音、视频效果实现了 IAMTimelineEffect、IAMTimelineSplittable；

过渡实现了 IAMTimelineTrans、IAMTimelineSplittable。

提示：组也是一种复合节点，所以它也实现了 IAMTimelineComp 接口。

比如我们现在要创建一个媒体源对象，代码如下：

```
IAMTimelineObj * pSrc = NULL;
pTL->CreateEmptyNode(&pSrc, TIMELINE_MAJOR_TYPE_SOURCE);
```

然后就是在媒体源对象上进行各种参数设置。这些参数包括必需参数和可选参数。

必需参数如下：

- 调用 IAMTimelineObj::SetStartStop 设置时间线时间（注意：千万不要给用一个轨道上的不同媒体源设置重叠的时间线时间，否者会引起不确定的错误）；
- 如果媒体源是一个文件，则调用 IAMTimelineSrc::SetMediaName 设置这个文件路径；
- 调用 IAMTimelineSrc::SetMediaTimes，给媒体源设置媒体时间（如果媒体源是静态图片文件则不用设置）。

可选参数如下：

- 调用 IAMTimelineSrc::SetStretchMode 设置视频媒体源的缩放模式，即当源文件的图像大小与组上设置的输出大小不一致时，DES 内部采用的缩放算法。DES 定义的几种缩放模式有：
  - RESIZEF_CROP，不进行缩放。当源图像太小时，则将图像周围填成黑色；太大时，将超出的部分剪裁掉；
  - RESIZEF_PRESERVEASPECTRATIO，尽量缩放到组上设定的图像大小，且保持源图像的宽高比率。当图像的高按照宽度的缩放比例进行缩放后仍然不能

填满整幅图像，则上下各加黑边（LetterBox）；

- RESIZEF_PRESERVEASPECTRATIO_NOLETTERBOX，尽量缩放到组上设定的图像大小并填满整幅图像，且保持源图像的宽高比率。缩放后的图像宽、高超过组上设定的图像大小的，则被剪裁掉；
- RESIZEF_STRETCH，简单拉伸源图像，填满组上设定的图像大小的整幅图像。

如图 9.8 所示是各种缩放模式的效果图。

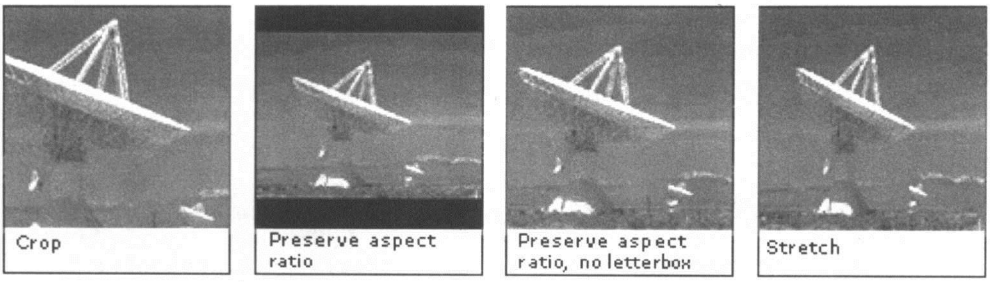

图 9.8　DES 中的 4 种图像缩放模式

提示

DES 默认情况下使用 RESIZEF_STRETCH 缩放模式。

- 调用 IAMTimelineSrc::SetMediaLength 设置媒体源的总时间长度，而且一般是在设置媒体开始、结束时间之前设置。这么做的好处是，当我们设置媒体时间时，DES 会检查结束时间的有效性（如果超出范围，则自动进行纠正）。我们可以使用 DES 提供的媒体探测器（Media Detector）组件来获知一个媒体文件的总时间长度；
- 调用 IAMTimelineSrc::SetStreamNumber 设置媒体源输出流的索引。默认情况下，DES 使用第一个符合组上设置的媒体类型的输出流。如果媒体源能够产生同样符合组上媒体类型的多个流，我们可能需要显式地选择其中一个。我们可以使用 DES 提供的媒体探测器组件来获知一个媒体文件有几个输出流。

媒体探测器这个组件的 CLSID 为 CLSID_MediaDet，实现的接口为 ImediaDet。它有两种工作模式。信息获取模式和抓图模式。当组件创建后，初始为信息获取模式；我们可以调用接口方法 EnterBitmapGrabMode、GetBitmapBits 或 WriteBitmapBits 进入抓图模式。但要注意，进入抓图模式之后，将不能再返回到信息获取模式。

信息获取模式下的一般使用方法是：

（1）使用 CoCreateInstance 函数创建一个 CLSID_MediaDet 组件实例；

（2）调用 IMediaDet::put_Filename 设置媒体文件路径；

（3）调用 IMediaDet::get_OutputStreams 获取输出流的个数（只统计媒体类型为 MEDIATYPE_Video 和 MEDIATYPE_Audio 的流）；

（4）调用 IMediaDet::put_CurrentStream 指定一个输出流，然后调用 IMediaDet::get_

StreamType 获知当前流的类型（视频或音频），调用 IMediaDet::get_ StreamLength 获得当前流的总时间长度，调用 IMediaDet::get_FrameRate 获取视频流输出帧率。

当媒体探测器工作在抓图模式时，一般也是首先在信息获取模式下选择一个想要抓取的输出流，然后调用 IMediaDet::GetBitmapBits 或 IMediaDet::WriteBitmapBits 抓取指定时刻的那一帧。使用 GetBitmapBits 抓图的代码如下：

```
long size;
// 首先获取图像帧的大小
hr = pDet->GetBitmapBits(0, &size, 0, width, height);
if (SUCCEEDED(hr))
{
    char *pBuffer = new char[size];
    if (!pBuffer)
        return E_OUTOFMEMORY;
    try {
        hr = pDet->GetBitmapBits(0, 0, pBuffer, width, height);
    }
    catch (...) {
        delete [] pBuffer;
        throw;
    }
    if (SUCCEEDED(hr))
    {
        BITMAPINFOHEADER *bmih = (BITMAPINFOHEADER*)pBuffer;
        HDC hdcDest = GetDC(0);

        // Find the address of the start of the image data
        void *pData = pBuffer + sizeof(BITMAPINFOHEADER);

        // Note: In general a BITMAPINFOHEADER can include extra color
        // information at the end, so calculating the offset to the image
        // data is not generally correct. However, the IMediaDet interface
        // always returns an RGB-24 image with no extra color information

        BITMAPINFO bmi;
        ZeroMemory(&bmi, sizeof(BITMAPINFO));
        CopyMemory(&(bmi.bmiHeader), bmih, sizeof(BITMAPINFOHEADER));
        HBITMAP hBitmap = CreateDIBitmap(hdcDest, bmih, CBM_INIT,
            pData, &bmi, DIB_RGB_COLORS);
    }
    delete[] pBuffer;
}
```

提示 使用 DES 的媒体探测器进行抓图，返回的图像帧总是 24 位像素格式的。与 5.8 节介绍的方法相比，DES 的这种方法也不失为一种抓图的好方法。

除了普通的媒体文件（.avi、.mpg 等）外，DES 的媒体源还支持如下文件：

- 静态图片文件：格式包括.bmp、.gif、.jpg、.tga 等。当媒体源处理静态文件时，我们一般需要调用 IAMTimelineSrc::SetDefaultFPS(0)，设置媒体源默认输出帧率为 0。
- DIB 图片序列（DIB Sequences）：在同一个目录下有一系列的图片文件（格式为.bmp、.jpg 或.tga 等），并且文件名按数字序号递增命名，如 Image001.bmp、Image002.bmp、Image003.bmp 等，DES 媒体源能够把它们当作是一组图片来处理。在这种情况下，我们只需要将第一个文件的文件名设置给媒体源，并且调用 IAMTimelineSrc::SetDefaultFPS 设置输出帧率。如果设定了输出帧率，在给定的时间线时间范围内，所有的图片都播放完了，则媒体源会输出纯黑颜色的图像帧。
- GIF 动画文件：这种动画文件有自己的输出帧率，所以我们没有必要再次设置。当 GIF 图像包含透明色时，DES 会将透明色转化为 RGB（0,0,0），而图像内容的像素值落在 RGB（0~7,0~7,0~7）范围内的都转化为 RGB（8,8,8）。为了重新透明效果，我们可以使用 DES 的色键过渡（Key Transition）对 RGB（0,0,0）进行抠像。
- 纯色视频源（Video Color Source）：这是一个 Source Filter，它的 CLSID 为 CLSID_ColorSource。作为媒体源时，我们需要通过接口方法 IAMTimelineObj::SetSubObjectGUID 将 CLSID_ColorSource 设置进去。工作时，它可以输出纯色的视频流，主要用于视频内容的淡进（Fade-in）和淡出（Fade-out）。
- 自定义的 Source Filter：我们可以开发自己的 Source Filter，将它作为 DES 的媒体源参与到视频编辑项目中去。比如为 DES 扩展视频标题叠加，我们就可以开发一个产生纯色背景下字幕流的 Source Filter，然后利用 DES 的色键过渡与主视频内容进行叠加。DES 对我们的这种自定义的 Source Filter 提出了要求：支持随机定位操作（即在输出 Pin 上实现 IMediaSeeking 接口）；输出 Pin 上产生的数据的媒体类型必须是 DES 支持的（非压缩格式或者 DirectShow 能够自动完成解码的压缩数据）；如果要在 Filter 上设置或获取属性，需要实现自动化接口（IDispatch）。

提示　自定义的 Source Filter 在 DES 中的使用方法与纯色视频源的使用方法类似。实际上，对于 DES 来说，它们是同一种类型的扩展组件。

9.1.4　音、视频效果与过渡

在介绍音视频效果与过渡之前，我们首先要来介绍一下 DES 中的“子对象”（Sub-object）概念。

有的 DES 对象并不做实际的工作，而是让特定的子对象去完成。子对象也是一种 COM 组件，我们可以将它的 GUID 通过 IAMTimelineObj::SetSubObjectGUID 设置给 DES 对象，并实际由 DES 对象创建、使用。音、视频效果与过渡就是属于这种情况，并且，纯色视频源和自定义 Source Filter 也是属于这种情况。

提示　DES 使用的音、视频效果与过渡处理子对象，都采用了 DirectX Transform Object 技术。只不过前者是一个输入一个输出的组件，而后者是两个输入一个输出的组

件。遗憾的是，新版本的 DirectX SDK 不再支持这种组件的开发（除非我们退回到 DirectX SDK 6.0）。现在我们能够直接使用的音、视频效果与过渡，只有 DES 本身提供的几种，还有就是微软公司的 IE 自带的几种。

使用音、视频效果与过渡的一般过程如下：

（1）调用 IAMTimeline::CreateEmptyNode 创建一个音、视频效果或过渡对象，参数指定为 TIMELINE_MAJOR_TYPE_EFFECT 或 TIMELINE_MAJOR_TYPE_TRANSITION；

（2）调用 IAMTimelineObj::SetSubObjectGUID 指定子对象；

（3）调用 IAMTimelineObj::SetStartStop 设置时间线时间；

（4）将音、视频效果或过渡对象加入到时间线中，一般是加在一个轨道中，加入音、视频效果调用 IAMTimelineEffectable::EffectInsBefore，加入过渡调用 IAMTimelineTransable::TransAdd；

（5）在音、视频效果或过渡对象上设置属性（使用 IPropertySetter 接口）。

使用 DES 自带的 SMPTE 过渡的代码如下：

```
IAMTimeline *pTL;
IAMTimelineTrack *pTrack;
// Create timeline with track (not shown)
// 创建过渡对象
IAMTimelineObj *pTransObj = NULL;
hr = pTL->CreateEmptyNode(&pTransObj, TIMELINE_MAJOR_TYPE_TRANSITION);
// 设置过渡子对象
hr = pTransObj->SetSubObjectGUID(CLSID_DxtJpeg);  // SMPTE Wipe
// 设置过渡对象的时间线时间（包括开始时间和结束时间）
hr = pTransObj->SetStartStop(50000000, 100000000);
// 将过渡对象插入到轨道中
IAMTimelineTransable *pTransable = NULL;
hr = pTrack->QueryInterface(IID_IAMTimelineTransable, (void
     **)&pTransable);
hr = pTransable->TransAdd(pTransObj);
pTransable->Release();
pTransObj->Release();
```

在 SMPTE 过渡对象上设置属性的代码如下：

```
IPropertySetter     *pProp;    // Property setter
IAMTimelineObj      *pTransObj;  // Transition object
// Create an SMPTE Wipe transition object (Not shown)
// 创建一个属性设置组件
hr = CoCreateInstance(CLSID_PropertySetter, NULL, CLSCTX_INPROC_SERVER,
    IID_IPropertySetter, (void**) &pProp);
// Error checking is omitted for clarity...
DEXTER_PARAM param;
DEXTER_VALUE *pValue =
(DEXTER_VALUE*)CoTaskMemAlloc(sizeof(DEXTER_VALUE));
// 参数初始化
```

```
param.Name = SysAllocString(L"MaskNum");
param.dispID = 0;
param.nValues = 1;
// Initialize the value
pValue->v.vt = VT_BSTR;
pValue->v.bstrVal = SysAllocString(L"120"); // Oval
pValue->rt = 0;
pValue->dwInterp = DEXTERF_JUMP;
pProp->AddProp(param, pValue);
// Free allocated resources
SysFreeString(param.Name);
VariantClear(&(pValue->v));
CoTaskMemFree(pValue);
// 在过渡对象上设置属性
pTransObj->SetPropertySetter(pProp);
pProp->Release();
```

DES已经做好了音、视频效果和过渡组件，见表9.1。

表9.1 DES的音、视频效果和过渡组件

过渡组件	CLSID	描述
合成器（Compositor）	CLSID_DxtCompositor	可以将两路视频进行合成，实现画中画功能
色键（Key）	CLSID_DxtKey	根据像素的RGB值、Alpha值、色调、亮度等进行两路视频的抠像叠加
SMPTE Wipe	CLSID_DxtJpeg	有100多种标准的过渡实现
Alpha设置器（Alpha Setter）	CLSID_DxtAlphaSetter	为图像设置Alpha值，可以配合色键过渡组件一起使用
音量调整器（Volume Envelope）	CLSID_AudMixer	支持动态的音量调整，可以实现一种音量的淡进淡出效果

除此之外，我们还可以使用系统中已经注册的第三方音、视频效果和过渡组件，这就要用到枚举。因为视频效果注册在CLSID_VideoEffects1Category类型目录下，过渡组件注册在CLSID_VideoEffects2Category类型目录下，我们只需分别枚举这两个目录，就能得到本机机器上注册的所有音、视频效果和过渡组件了。

提示　在SDK安装目录下的Samples\C++\DirectShow\Editing\TransViewer，提供了枚举系统中注册的过渡组件的完整演示。

最后，我们来看一下过渡的方向和进度问题。默认情况下，过渡的方向是从低优先级的层（Layer，如轨道、复合节点）转到高优先级的层。我们也可以调用IAMTimeline-Trans::SetSwapInputs来将这个方向反过来。另外，大部分过渡组件都有一个Progress属性，表示当前的合成进度（不同的进度决定了两路视频的不同合成比例）。默认情况下，进度总是从0慢慢转到1。

如图9.9所示，假设现在有一个过渡对象加在轨道1上，默认情况下，最终视频内容

表现为从轨道 0 渐渐过渡到轨道 1。如果翻转过渡方向后，我们将看到从轨道 1 到轨道 0 的过渡效果。如果在默认过渡方向的基础上翻转进度（即 Progress 值从 1 慢慢转到 0），我们看到的好像是从轨道 1 到轨道 0 的过渡。如果我们同时翻转过渡方向和进度，我们看到的也将好像是从轨道 0 过渡到轨道 1。

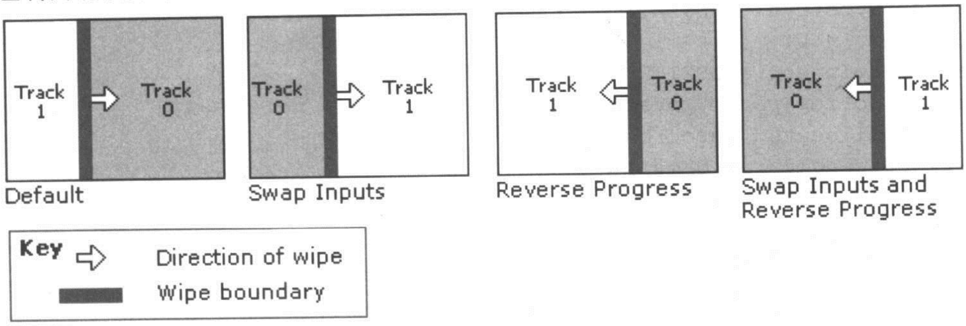

图 9.9　过渡的方向和进度

9.1.5　输出控制引擎

时间线对象维持的只是一个编辑信息集合，那么谁来负责将这些编辑信息“翻译”成 DirectShow Filter 架构来最终实现呢？DES 使用了输出控制引擎（Render Engine）。

DES 提供了两种控制引擎：基本控制引擎（Basic Render Engine）和智能控制引擎（Smart Render Engine）。前者输出非压缩格式的数据，主要用于预览（当然也可用于输出到文件保存）；后者输出压缩格式的数据，主要用于写文件（当然也可用于预览，但必须先将压缩格式的数据解码，效率比较低）。

基本控制引擎的设计思路是，首先将各个媒体源解码成统一的一种非压缩格式，并将它的输出接入到 DES 的一个控制 Filter 上；这个控制 Filter 再根据时间线对象描述的编辑信息，对各路输入的数据进行处理，最后输出一个合成流。而智能控制引擎的设计，完全是出于提高性能考虑（特别是对于写文件的应用）。我们可以设置一个压缩格式作为最终的输出格式，当媒体源的原始格式与这个输出格式相同时，智能控制引擎就不会对媒体源构建一条解码后重压缩的 Filter 链路，而是直接输出压缩数据。注意，智能控制引擎只支持视频的智能重压缩。

提示　构建媒体源的解码链路时，DES 内部使用了智能连接方法，这样可以尽可能地利用本地系统中注册的第三方 Filter。但同时，通过智能连接构建的 Filter Graph 也是不稳定的，因为用这种方法构建的 Filter Graph 具有不确定性（我们不知道目标系统中有什么 Filter，是否有搞“恶意”破坏的 Filter，也不知道构建完成的具体 Filter Graph 到底使用了哪些 Filter）。因此，直接基于 DES 开发的应用程序的健壮性并不能得到保障。

输出控制引擎构建 Filter Graph 时，一般分为两步。首先是构建一个前端（Front End），如图 9.10 所示。

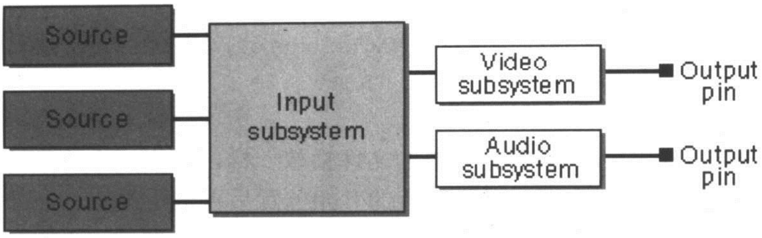

图 9.10　DES Filter Graph 的前端构建

我们看到，前端构建成功后，典型的情况下会有一个视频的输出 Pin 和一个音频的输出 Pin。它们等待后续的不同用途的连接。比如我们要进行预览时，分别接上一个 Renderer Filter，如图 9.11 所示。

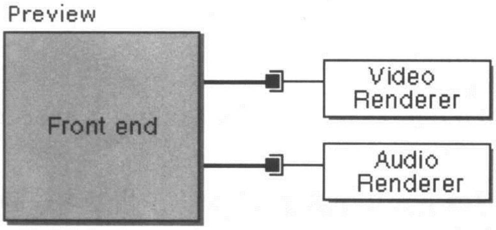

图 9.11　DES Filter Graph 的前端预览

如果我们想将最终的编辑作品生成一个媒体文件，如图 9.12 所示。

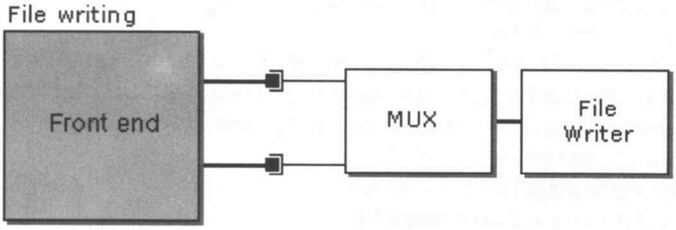

图 9.12　DES Filter Graph 的前端输出到文件

预览一个 DES 编辑项目的代码（使用基本控制引擎）如下：

```
IRenderEngine *pRender = NULL;
hr = CoCreateInstance(CLSID_RenderEngine, NULL,
     CLSCTX_INPROC_SERVER, IID_IRenderEngine, (void**)&pRender);
// 设置时间线对象给控制引擎
hr = pRender->SetTimelineObject(pTL);
// 完成前端 Filter Graph 的构建
hr = pRender->ConnectFrontEnd();
// 将前端构建完成的输出 Pin 分别接到 Renderer Filter
hr = pRender->RenderOutputPins();
IGraphBuilder    *pGraph = NULL;
IMediaControl    *pControl = NULL;
// 得到控制引擎创建的 Filter Graph Manager 指针
hr = pRender->GetFilterGraph(&pGraph);
```

```
// 从 Filter Graph Manager 上获取 Filter Graph 的控制接口
hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
// 运行这个 DES 创建的 Filter Graph
hr = pControl->Run();
```

当一遍播放完成后，我们可以跟普通的文件播放一样，使用 Filter Graph Manager 上获得的 IMediaSeeking 接口，重新定位到流的开始位置后再进行播放。但如果我们修改了时间线上的编辑信息，我们需要如下操作：

```
// 重新调用前端构建函数
// 如果返回值为 S_WARN_OUTPUTRESET，则需要再次调用 RenderOutputPins
hr = pRender->ConnectFrontEnd();
if (hr == S_WARN_OUTPUTRESET)
{
    hr = pRender->RenderOutputPins();
}
LONGLONG llStart = 0;
// 重新定位到开始的地方再运行
hr = pSeek->SetPositions(&llStart, AM_SEEKING_AbsolutePositioning, 0, 0);
hr = pControl->Run();
```

将 DES 编辑项目最终输出到文件保存的代码（使用基本控制引擎）如下：

```
IRenderEngine *pRender = NULL;
hr = CoCreateInstance(CLSID_RenderEngine, NULL, CLSCTX_INPROC,
    IID_IRenderEngine, (void**) &pRender);
hr = pRender->SetTimelineObject(pTL);
hr = pRender->ConnectFrontEnd( );
// 创建一个 Capture Graph Builder 组件，辅助 Filter Graph 的构建
CoCreateInstance(CLSID_CaptureGraphBuilder2, NULL, CLSCTX_INPROC,
    IID_ICaptureGraphBuilder2, (void **)&pBuilder);
IGraphBuilder *pGraph;
// 得到 DES 控制引擎创建的 Filter Graph
pRender->GetFilterGraph(&pGraph);
// 初始化 Capture Graph Builder 组件
pBuilder->SetFiltergraph(pGraph);
// 创建一个 AVI 输出文件
IBaseFilter *pMux;
pBuilder->SetOutputFileName(&MEDIASUBTYPE_Avi,
    OLESTR("Output.avi"), &pMux, NULL);
// 得到时间线总共包含的组的个数（一个组会对应一个前端输出 Pin）
long NumGroups;
pTimeline->GetGroupCount(&NumGroups);
// Loop through the groups and get the output pins
for (i = 0; i < NumGroups; i++)
{
    IPin *pPin;
    if (pRender->GetGroupOutputPin(i, &pPin) == S_OK)
    {
        IBaseFilter *pCompressor;
        // Create a compressor filter (Not shown.)
```

```
            // Set compression parameters (Not shown.)
            // Connect the pin
            pBuilder->RenderStream(NULL, NULL, pPin, pCompressor, pMux);
            pCompressor->Release();
            pPin->Release();
        }
    }
    // 然后从 Filter Graph Manager 上获得控制接口进行控制
    // ...
```

使用智能控制引擎将 DES 编辑项目输出到文件保存时，步骤与使用基本控制引擎时类似，只不过我们现在将压缩器放到了前端，并根据实际的需要才进行解码后重压缩。

提示 不要使用智能控制引擎读写 Windows Media 格式的文件。

我们可以使用 IAMTimelineGroup::SetSmartRecompressFormat 在视频组上设置压缩的媒体类型（调用这个接口方法前，也必须调用 IAMTimelineGroup::SetMediaType 设置一个非压缩的媒体类型），参数使用了 SCompFmt0 数据结构，代码如下：

```
typedef struct _SCompFmt0
{
    long nFormatId;
    AM_MEDIA_TYPE MediaType;
} SCompFmt0;
```

其中，nFormatId 保留为 0。实现的代码如下：

```
SCompFmt0 *pFormat = new SCompFmt0;
memset(pFormat, 0, sizeof(SCompFmt0));
pFormat->nFormatId = 0;
// Initialize pFormat->MediaType (Not shown.)
pGroup->SetSmartRecompressFormat( (long*) pFormat );
```

值得注意的是，所谓组上设置的输出压缩格式与媒体源的原始压缩格式相同，除了指格式本身相同外，还包括视频图像的宽、高、像素位数、帧率等属性相同。（我们可以使用 DES 提供的媒体探测器的接口方法 IMediaDet::get_StreamMediaType 获取源文件的媒体类型作为参考。）

使用智能控制引擎还需要注意的事项有：

- 创建控制引擎时使用 CLSID_SmartRenderEngine；
- 创建前端完成后，但在连接前端输出 Pin 之前设置压缩格式，方法是调用视频组上的 ISmartRenderEngine::GetGroupCompressor 方法获得压缩器 Filter，然后再得到 IAMVideoCompression 接口进行设置；
- 在我们连接前端的输出 Pin 时，不再需要压缩器，因为我们输出的已经是压缩格式了。

9.1.6 错误日志

对于 DES 内部产生的错误，我们可以使用一种回调机制来处理。也就是说，我们要在应用程序中实现一个带有 IAMErrorLog 接口的 COM 组件，然后通过时间线对象上的接口方法 IAMSetErrorLog::put_ErrorLog 设置进去。当 DES 内部有错误发生时，会调用 IAMErrorLog::LogError 进行“汇报”。

我们在应用程序中实现这个 COM 组件（假设 COM 类为 CErrReporter），代码如下：

```
class CErrReporter : public IAMErrorLog
{
protected:
    long    m_lRef; // Reference count
public:
    CErrReporter() { m_lRef = 0; }
    // IUnknown
    STDMETHOD(QueryInterface(REFIID, void**));
    STDMETHOD_(ULONG, AddRef());
    STDMETHOD_(ULONG, Release());
    // IAMErrorLog
    STDMETHOD(LogError(LONG, BSTR, LONG, HRESULT, VARIANT*));
};
```

实现 IUnknown 的 3 个接口方法的代码如下：

```
STDMETHODIMP CErrReporter::QueryInterface(REFIID riid, void **ppv)
{
    if (ppv == NULL) return E_POINTER;

    *ppv = NULL;
    if (riid == IID_IUnknown)
        *ppv = static_cast<IUnknown*>(this);
    else if (riid == IID_IAMErrorLog)
        *ppv = static_cast<IAMErrorLog*>(this);
    else
    return E_NOINTERFACE;
    AddRef();
    return S_OK;
}
STDMETHODIMP_(ULONG) CErrReporter::AddRef()
{
    return InterlockedIncrement(&m_lRef);
}
STDMETHODIMP_(ULONG) CErrReporter::Release()
{
    // Store the decremented count in a temporary variable
    ULONG uCount = InterlockedDecrement(&m_lRef);
    if (uCount == 0)
    {
        delete this;
    }
```

```
    // Return the temporary variable, not the member
    // variable, for thread safety.
    return uCount;
}
```

DES 错误汇报回调函数 IAMErrorLog::LogError 的实现代码如下：

```
STDMETHODIMP CErrReporter::LogError(
    LONG Severity,             // Reserved. Do not use
    BSTR ErrorString,          // Description
    LONG ErrorCode,            // Error code
    HRESULT hresult,           // HRESULT that caused the error
    VARIANT *pExtraInfo)     // Extra information about the error
{
// 获得错误代码 ErrorCode
// …
// 获得错误描述字符串
// …

// 获取描述错误的额外信息
if( pExtraInfo )
{
    printf("\tExtra info: ");
    if( pExtraInfo->vt == VT_BSTR )       // Extra info is a BSTR
    {
        UINT len = SysStringLen(pExtraInfo->bstrVal);
        char *szExtra = new char[len];
        if (szExtra != NULL)
        {
            // Note - If the BSTR contains embedded NULL characters,
            // this will only pick up the first sub-string
            WideCharToMultiByte(CP_ACP, 0, pExtraInfo->bstrVal, -1,
                szExtra, len, 0, 0);
            printf("%s\n", szExtra);
            delete [] szExtra;
        }
    }
    else if( pExtraInfo->vt == VT_I4 )    // Extra info is an integer
        printf("%d\n", pExtraInfo->lVal);
    else if( pExtraInfo->vt == VT_R8 )    // Extra info is floating-point
        printf("%f\n", pExtraInfo->dblVal);
}
}
```

提示

我们在应用程序中实现 IAMErrorLog::LogError 函数时，不要使用阻塞调用的函数，应该尽量让函数体比较简单，并且处理完成后尽快返回。

使用这个错误日志组件的代码如下：

```
IAMSetErrorLog  *pSetLog = NULL;
IAMErrorLog     *pLog = new CErrReporter();
// 在时间线对象上获得 IAMSetErrorLog 接口
pTL->QueryInterface(IID_IAMSetErrorLog, (void **)&pSetLog);
// 将错误日志组件对象指针设置给时间线对象
pSetLog->put_ErrorLog(pLog);
pSetLog->Release();
```

9.1.7 DES 项目管理

时间线对象上维持的是各种编辑信息，我们可以将这些编辑信息保存到 XML 格式的文件中，也可以根据 XML 文件恢复一个时间线对象。这时候，我们用到了 XML Parser 组件。

XML Parser 组件的 CLSID 为 CLSID_Xml2Dex，它上面实现了一个 IXml2Dex 接口。我们可以使用 IXml2Dex::ReadXMLFile 读取一个 XML 格式的时间线描述文件（通常扩展名为.xtl），也可以使用 IXml2Dex::WriteXMLFile 将时间线对象内的编辑结构信息保存为一个 XML 格式的文件。

装载一个.xtl 文件的代码如下：

```
HRESULT         hr;
IAMTimeline     *pTL = NULL;
IXml2Dex        *pXML = NULL;
// 创建一个空的时间线对象
hr = CoCreateInstance(CLSID_AMTimeline, NULL, CLSCTX_INPROC_SERVER,
            IID_IAMTimeline, (void**)&pTL);
// 创建一个 XML Parser 组件
hr = CoCreateInstance(CLSID_Xml2Dex, NULL, CLSCTX_INPROC_SERVER,
            IID_IXml2Dex, (void**)&pXML);
BSTR bstrFile = SysAllocStringLen(OLESTR("C:\\example.xtl"), 15);
// 装载 XML 格式的文件，并将编辑信息转化为时间线对象的内部结构
hr = pXML->ReadXMLFile(pTL, bstrFile);
SysFreeString(bstrFile);
pXML->Release();
```

DES 编辑项目的管理还有其他要点，总结如下：

- 当我们已经将时间线对象转化为 Filter Graph 实现后，如果又改变了时间线编辑信息，我们需要再次调用 IRenderEngine::ConnectFrontEnd 进行前端构建；如果这个函数返回值为 S_WARN_OUTPUTRESET，则我们还必须重新构建前端输出 Pin 后续的 Filter 链路；
- 如果我们想要删除时间线对象中的所有编辑信息，调用 IAMTimeline::ClearAllGroups；
- 当我们不再想使用控制引擎进行 DES 项目的输出时，一般首先释放控制引擎创建的 Filter Graph 上的各个控制接口，然后调用控制引擎对象上的 IRenderEngine::ScrapIt；
- 控制引擎并不会对时间线对象增加引用计数，所以当控制引擎正在工作时，不要

释放时间线对象；

- 如果已经释放了时间线对象，那么不要再使用这个时间线对象中的其他任何 DES 对象（即使我们在外部已经保持了一个引用计数）；
- 如果我们创建了几个时间线对象，则要注意它们中的 DES 对象是不能相互交换的。记住，时间线中的 DES 对象只能是通过它自身的接口方法 IAMTimeline::CreateEmptyNode 来创建的；
- 不要用一个控制引擎去同时控制多个时间线对象的输出；
- 一个时间线对象可以被多个控制引擎使用，但不能同时进行，一般我们在要使用新的控制引擎之前会把老的删除掉（典型的情况，我们使用基本控制引擎预览 DES 编辑项目，而当输出写文件时改用智能控制引擎）；
- 当我们将时间线对象上的编辑信息保存到 XML 格式的文件时，Filter Graph 上的设置并不会保存（如使用智能控制引擎时设置的压缩格式、参数等）。这部分信息我们必须在应用程序中另外想办法保存，并且在装载 XML 文件生成相应的时间线对象后，特别地去恢复这些设置。

9.2　DES 剖　析

我们已经知道，DES 是在 DirectShow Filter 之上的一层增强应用。为了更进一步地理解 DES，我们下面将从 Filter Graph 这个层次着手，揭开 DES 的神秘面纱。

如图 9.13 所示为一个典型的用 DES 的控制引擎构建的 Filter Graph。

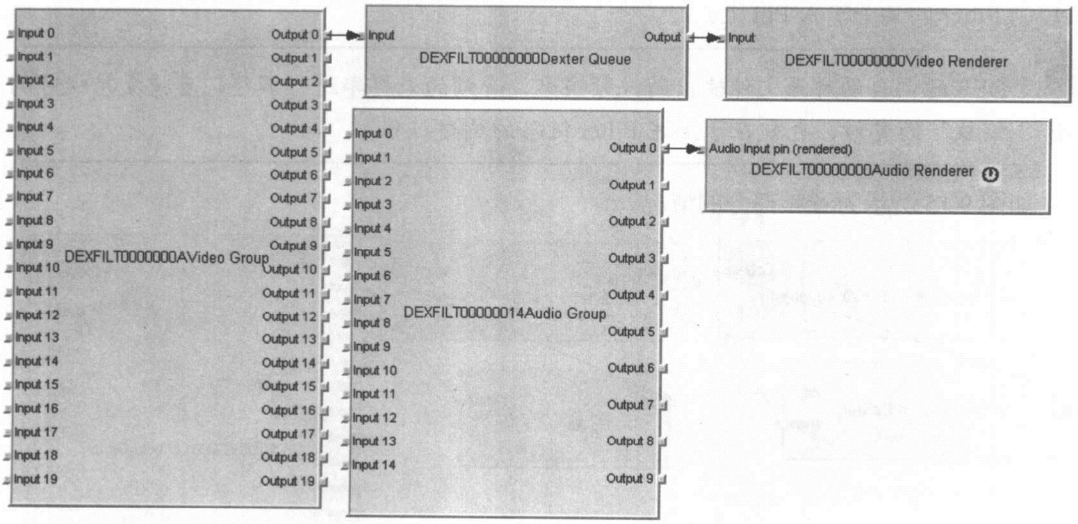

图 9.13　DES 的控制引擎构建的 Filter Graph

图中有两个 Filter 的外形很大，一个含有 20 个输入 Pin 和 20 个输出 Pin，另一个含有 15 个输入 Pin 和 10 个输出 Pin。两个 Filter 的输入输出 Pin 个数均是动态的。前者是 DES 视频部分的控制 Filter，后者是 DES 音频部分的控制 Filter。这两个 Filter 也就是 DES 的核

心所在。它们知道用户编辑的时间线结构，并且它们能够在 Filter Graph 运行的时候创建各个媒体源的解码链路。一个典型的媒体源的解码链路如图 9.14 所示。

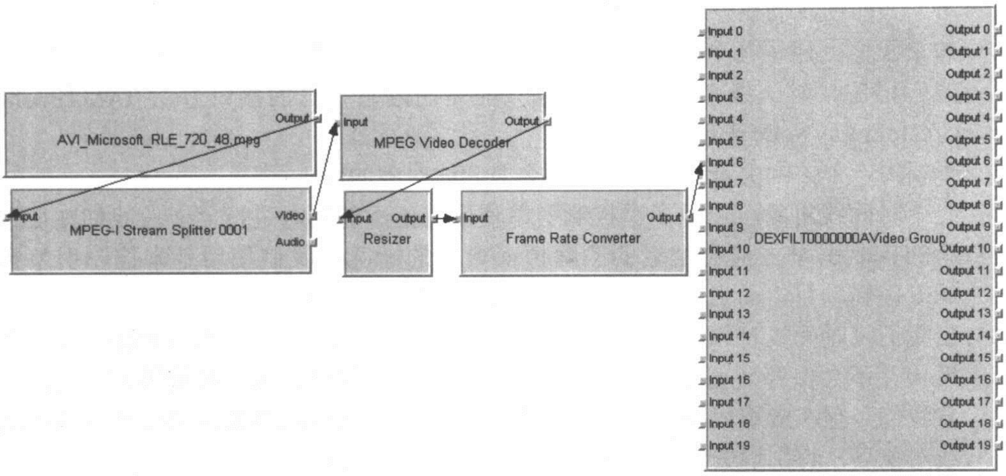

图 9.14　DES 创建的媒体源视频解码链路

从图中可以看出，我们加入了一个 MPEG1 的文件源，DES 会自动给这个文件进行解码链路的构建。首先经过 Splitter Filter 进行音、视频分离（因为这个文件源是加在视频组中的，所以 DES 只取 Splitter Filter 输出的视频流部分）；经过 Resizer（微软公司的一个没有公布的 Filter，用于视频图像的缩放）；然后再经过 Frame Rate Converter（这个 Filter 也是一个微软公司没有公布的 Filter，用于调整视频的输出帧率）；最后连接到 DES 的视频控制 Filter 的某个输入 Pin 上。

提示　DES 的“自动对源文件输出的视频帧率、音频的采样率进行调整，直接支持视频的缩放”的奥妙，也就在于上述 Filter 链路的构建过程。

如图 9.15 所示为音频部分的情况。

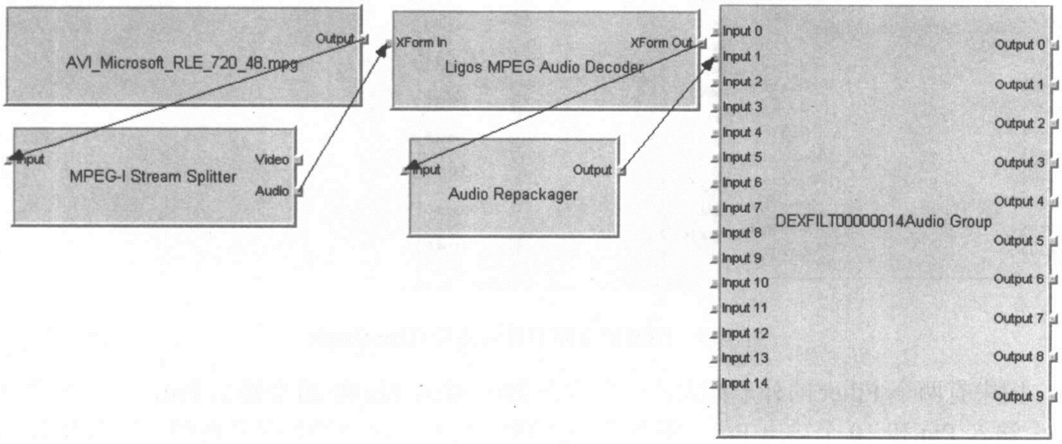

图 9.15　DES 创建的媒体源音频解码链路

从图中可以看出，音频解码链路与视频部分的解码链路是相似的。另外，我们还发现了一个新的 Filter：Audio Repackager。这个 Filter 负责将音频数据按设定的每秒钟输出多少个 Sample 重新进行打包，类似于视频部分的调整输出帧率的 Frame Rate Converter。

从理论上来说，DES 支持任何格式的媒体文件，只需系统中注册了对这种格式文件的解码 Filter 即可。如图 9.16 所示为 DES 对静态图片的支持。

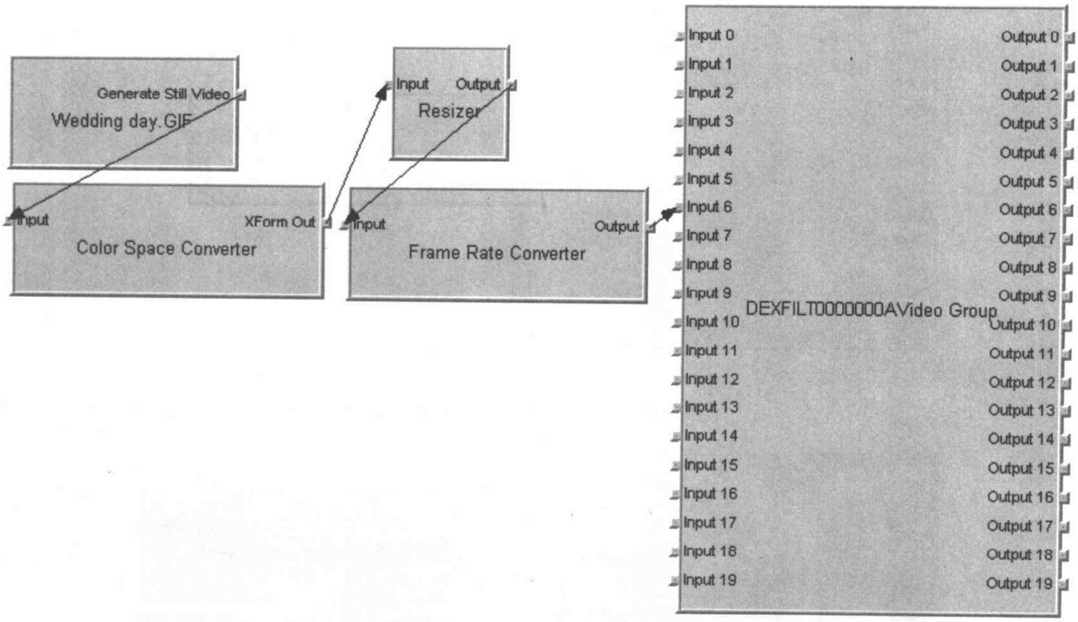

图 9.16　DES 对静态图片的解码链路

接下去，我们再来看一下 DES 对音、视频效果与过渡处理的实现。我们知道，DES 并没有直接实现这两部分功能，而是在时间线中建立了一个包装，把具体的工作交给了所谓的子对象去做。这个子对象就是 DirectX Transform Object（也是一种 COM 组件）。

如图 9.17 和图 9.18 所示，音、视频效果与过渡处理都使用了包装 Filter（Wrapper Filter）。我们发现，这两个包装 Filter 的 CLSID 是相同的（也就是说是同一种包装 Filter 的两个不同实例），不同的是实现音、视频效果的 Filter 是一进一出，而实现过渡处理的是两进一出。我们还可以看到，视频控制 Filter 的输出 Pin 经过包装 Filter 处理后，重新作为视频控制 Filter 的一个输入。这样的设计并不多见！

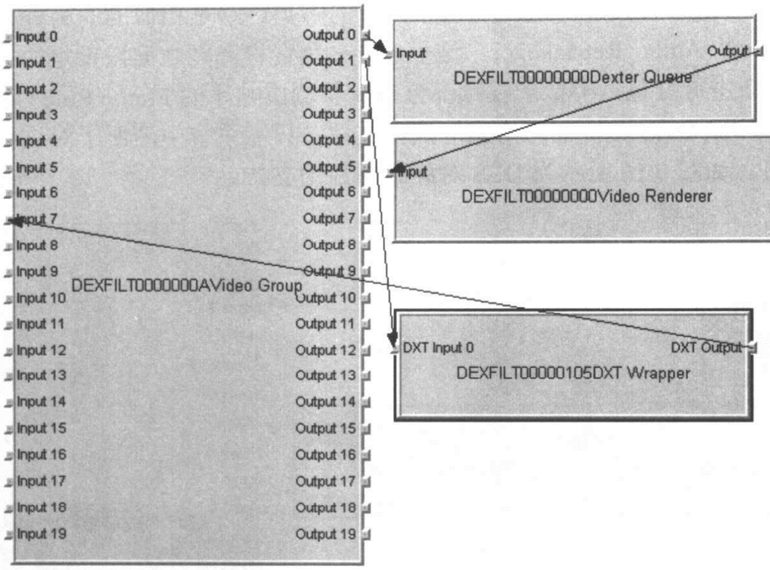

图 9.17　DES 对音视频效果的实现

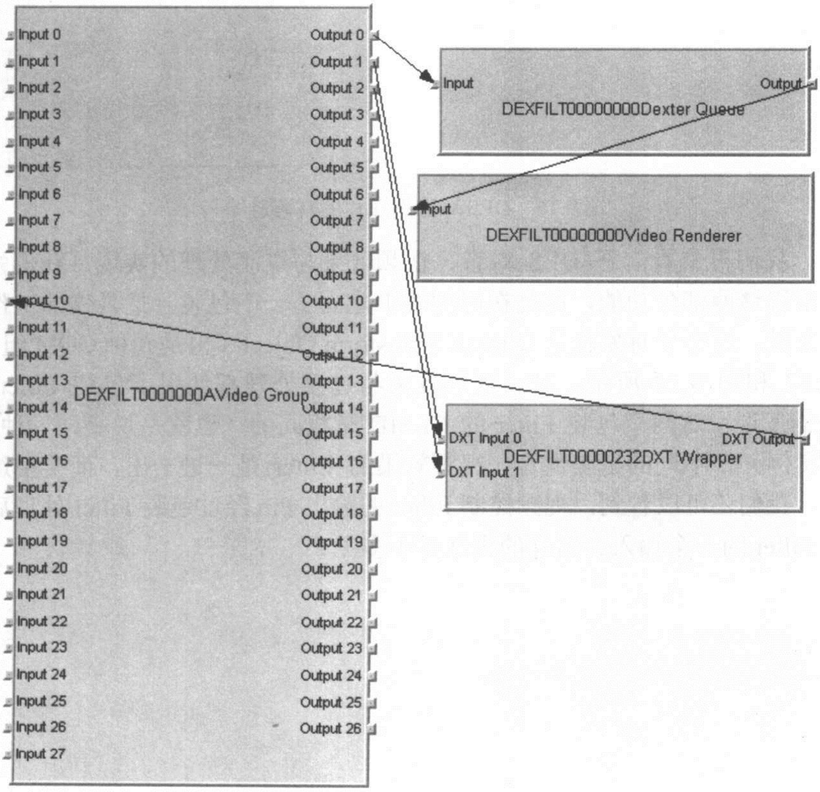

图 9.18　DES 对过渡处理的实现

接下去，我们再来看一下 DES 对音频合成的实现。大家知道，典型的时间线会有一

个视频组和一个音频组。如果整个音频组没有一个媒体源，或者一个 10s 的时间线中一个音频文件仅能提供 0~5s 的输出，DES 是如何处理的呢？

如图 9.19 所示，DES 会自动创建一个 Source Filter，它产生静音的 PCM 数据。这样就保证了音频组在整个时间线过程中肯定是有数据输出的。

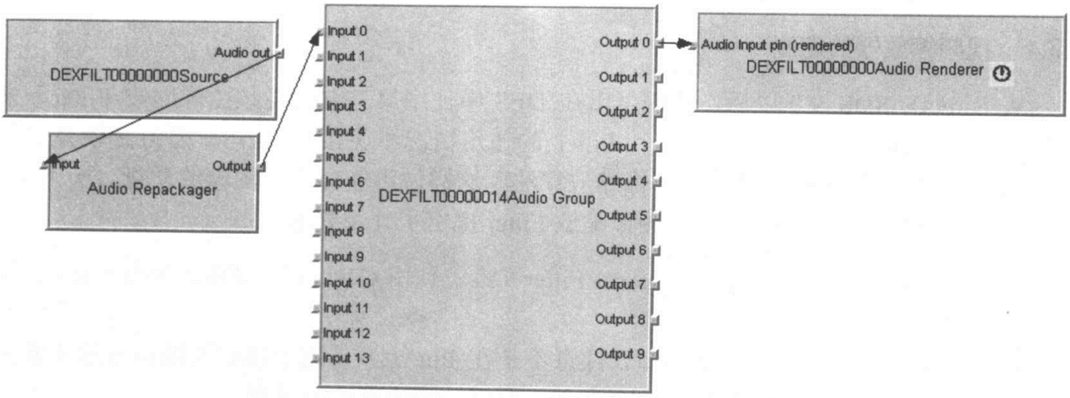

图 9.19　DES 的静音实现

下面我们再来看音频合成的一个例子。如图 9.20 所示，DES 的音频控制 Filter 将所有输入的 PCM 数据做简单的混合后输出。

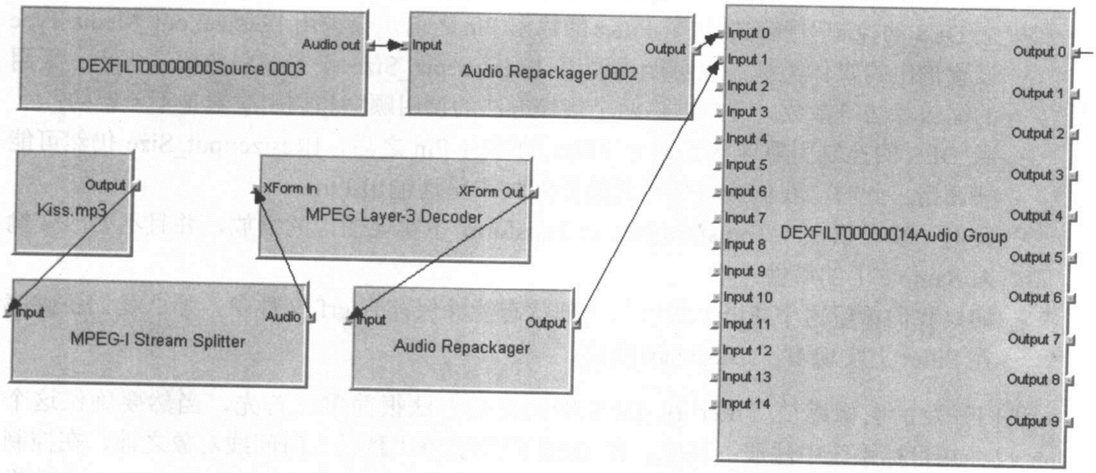

图 9.20　DES 对音频合成的实现

值得注意的是，当 DES 的控制引擎最初创建 Filter Graph 的时候，我们一般仅仅能够看到 DES 的那两个控制 Filter，还有就是音、视频效果与过渡处理的包装 Filter（如果有的话），以及 Audio Renderer 和 Video Renderer。当 Filter Graph 开始运行时，控制 Filter 才开始创建时间线上第一个媒体源的解码链路；而当 Filter Graph 运行到最后的时候，整个 Filter Graph 结构看起来可能是非常复杂的。（整个过程中，DES 能够根据时间线对象的描述进行 Filter Graph 的动态创建、连接。）

9.3 DES 新特性

与 DirectX SDK 以前的版本相比，DirectX SDK 9.0 对 DES 部分增加了几个新的特性，它们包括定制视频缩放器和选择解码器。

9.3.1 视频缩放器的定制

从 DirectX SDK 9.0 以后，我们可以在 DES 中使用第三方的视频图像缩放 Filter，来替换微软公司的 Resizer Filter（这个 Filter 的使用情况参见图 9.14）。因为微软公司的 Resizer Filter 默认使用简单拉伸的图像缩放算法，这样做可能会影响图像的质量。

不过，DES 也对这种第三方的图像缩放 Filter 提出了开发要求：

- 这个 Filter 必须是一个 Transform Filter（推荐使用 CTransformFilter 作为 Filter 的父类）；
- 在 Filter 上实现 IResize 接口（注意不是在 Pin 上），这个接口的接口方法主要完成 Filter 输出的媒体类型、图像大小、采用的缩放算法的设置；
- Filter 只能接受使用 FORMAT_VideoInfo 格式的媒体类型；
- Filter 应该能够支持所有非压缩的 RGB 格式输入（包括带有 Alpha 通道的 MEDIASUBTYPE_ARGB32），可以拒绝接受格式描述中（BITMAPINFOHEA-DER 数据结构的成员）biHeight 小于 0 的媒体类型；
- 在 DES 的控制引擎连接这个 Filter 的输出 Pin 之前，会调用 IResize::put_Media-Type 设置输出的媒体类型，也可能再调用 IResize::put_Size 设置输出的图像大小、采用的缩放算法等参数。注意，这两个接口方法的调用顺序和调用次数是不一定的；
- 当 DES 的控制引擎连接了这个 Filter 的输出 Pin 之后，IResize::put_Size 仍然可能被调用。此时，我们应该使用新的图像大小重连输出 Pin；
- 我们一般可以在 CTransformFilter::Transform 中实现图像的缩放，并且不要改变输入 Sample 上的属性；
- 如果我们希望这个 Filter 的一些内部状态能够保存到.grf 文件中，那么我们还应该在 Filter 上实现 IPersistStream 接口。

我们开发的图像缩放 Filter 在 DES 中的使用方法很简单。首先，当然要确保这个 Filter 已经在目标系统中注册；其次，在 DES 控制引擎“翻译”时间线对象之前，在控制引擎对象上获得 IRenderEngine2 接口，然后调用 IRenderEngine2::SetResizerGUID，参数设为我们的 Filter 的 CLSID。

9.3.2 解码器的选择

我们知道，DES 使用智能连接方法构建媒体源的解码链路（典型的情况发生在控制引擎上的接口方法 IRenderEngine::ConnectFrontEnd 被调用时）。我们也已经提到过，这种方法构建的 Filter 链路带有不确定性，有时候还可能会影响到应用程序的健壮性。DirectX SDK 9.0 提供了一种新的回调机制，让应用程序有机会影响到这个“智能”过程。

对于解码一种媒体格式，系统中可能存在多个这样的 Filter。我们可能希望 DES 能够

使用我们指定的 Filter。不过，就目前来说，我们还没有办法直接做到这一点。于是，DirectShow 提供了一种“间接”的方法，即将通过智能连接算法选中的 Filter，都通过一种回调机制让应用程序进行一下确认；应用程序可以接受，也可以拒绝。使用这种回调机制的关键，就是应用程序上一个带有 IAMGraphBuilderCallback 接口的组件实现。

IAMGraphBuilderCallback 接口有如下两个接口方法：

- SelectedFilter：当智能连接算法选中一个 Filter 但还没有创建它时调用这个函数。
- CreatedFilter：当智能连接算法选中的 Filter 已经创建了但还没连接之前调用这个函数。

下面，我们就来介绍这个回调机制在应用程序上的实现方法。首先声明一个 COM 类 GraphBuilderCB，并实现 IUnknow 和 IAMGraphBuilderCallback 的各个接口方法，代码如下：

```
class GraphBuilderCB : public IAMGraphBuilderCallback
{
public:
    // Method declarations (not shown).
};
```

然后就是这个组件的使用问题。我们可以创建一个 Filter Graph Manager，并从它上面获得 IObjectWithSite 接口，再调用接口方法 IObjectWithSite::SetSite 将我们的回调组件设置进去。于是，一个受我们应用程序“监控”的 Filter Graph 产生了。代码如下：

```
// Declare an instance of the callback object
GraphBuilderCB GraphCB;
// Create the Filter Graph Manager
CComPtr<IGraphBuilder> pGraph;
hr = pGraph.CoCreateInstance(CLSID_FilterGraph);
if (FAILED(hr))
{
    // Handle error (not shown)
}
// Register to receive the callbacks
CComQIPtr<IObjectWithSite> pSite(pGraph);
if (pSite)
{
    hr = pSite->SetSite((IUnknown*)&GraphCB);
}
```

随后，我们就要让 DES 的控制引擎在我们上面创建的 Filter Graph 中构建具体的 Filter 链路。还好，DES 提供了一个接口方法 IRenderEngine::SetFilterGraph 来设置，代码如下：

```
CComPtr<IRenderEngine> pRender;
hr = pRender.CoCreateInstance(CLSID_RenderEngine);
if (FAILED(hr))
{
    // Handle error (not shown)
}
hr = pRender->SetFilterGraph(pGraph);
```

我们再来讨论一下在回调函数中，实现解码 Filter“筛选”的策略（我们一般在 IAMGraphBuilderCallback::SelectedFilter 中实现）：

- 在实际的 Filter Graph 构建之前，调用接口方法 IFilterMapper2::EnumMatchingFilters，枚举得到系统中解码特定压缩格式的所有 Filter；
- 在枚举得到的 Filter 标识对象上得到它的 DisplayName 属性；
- 将枚举得到的 Filter 的 DisplayName 生成一个列表，但排除掉我们希望 DES 使用的解码 Filter 的 DisplayName。一般 DisplayName 的格式为：OLESTR("@device:sw:{CategoryGUID}\{FilterCLSID}")。其中，CategoryGUID 为 Filter 注册的类型目录，FilterCLSID 是 Filter 的 CLSID，它们都是一串 GUID 字符。对于 DMO 组件，上述的 sw 应该是 dmo。
- SelectedFilter 函数被调用时，参数是当前被智能连接算法选中的 Filter 标识对象指针，我们可以得到它的 DisplayName，然后与我们上面生成的列表进行比较。如果在列表内，则让 SelectedFilter 返回一个错误值拒绝这个 Filter；如果不在列表内，我们返回 S_OK 表示接受这个 Filter。

提示

下载地址（见前言）对应部分提供了一个 DESCallback 例子，演示了 IAMGraphBuilderCallback 接口的一种实现、使用的方法，可供读者参考。

9.4 实 例 解 剖

DirectShow SDK 提供了几个 DES 应用的例子，我们这里只重点介绍一个关于如何编辑时间线对象并预览的例子，它位于 SDK 安装目录的 Samples\C++\DirectShow\Editing\TimelineTest 下。

9.4.1 实现的功能

如图 9.21 所示是 TimelineTest 例子的程序界面。这是一个类似于控制台的程序，执行流程是：WinMain | CoInitializeEx | TimelineTest | CoUninitialize。主要演示了两个视频文件和两个音频文件在时间线上的编排，以及加入音、视频效果与过渡处理等的方法。

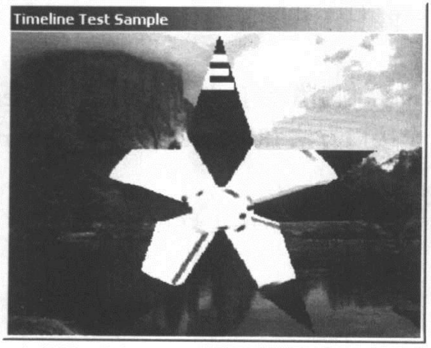

图 9.21 TimelineTest 例子的程序界面

9.4.2 实现要点

TimelineTest 例子的时间线结构参考如图 9.22 所示。

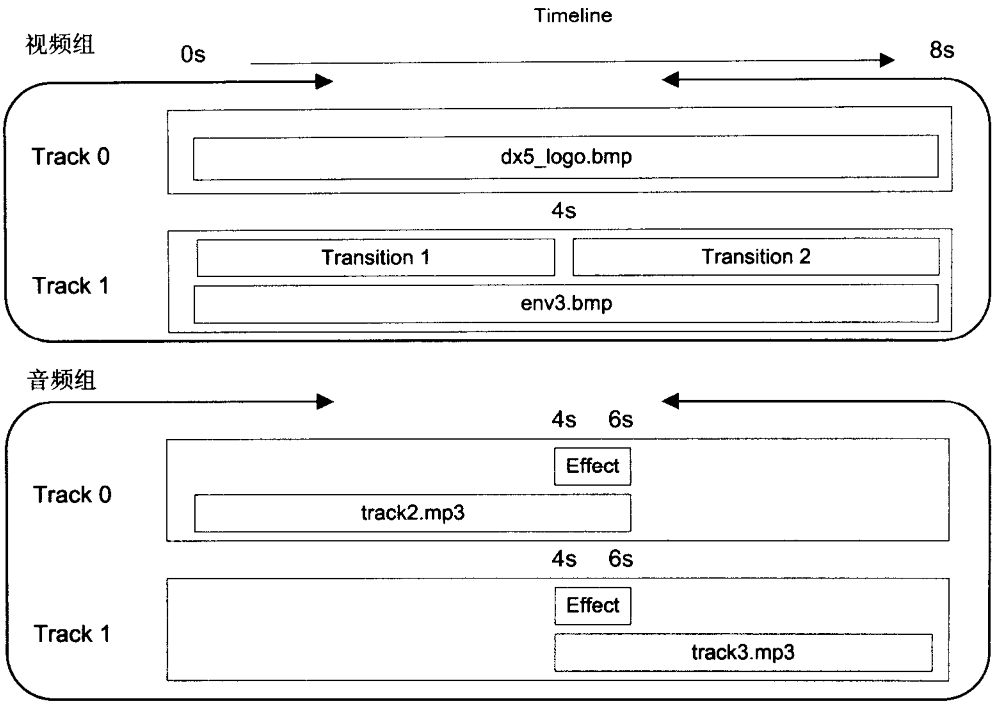

图 9.22 TimelineTest 例子的时间线结构

通过 TimelineTest 例子的学习，我们将掌握如下知识点：

- 创建一个事件线对象；
- 创建组，并标识为视频组或音频组；
- 在轨道上加入视频过渡处理，以及过渡子对象的设置；
- 在轨道上加入音频效果，并实现动态属性的配置；
- 使用控制引擎。

创建一个事件线对象

这是在做任何 DES 编辑之前最应先做的事情，代码如下：

```
CComPtr< IAMTimeline >    pTimeline;
hr = CoCreateInstance(
                            CLSID_AMTimeline,
                            NULL,
                            CLSCTX_INPROC_SERVER,
                            IID_IAMTimeline,
                            (void**) &pTimeline
                            );
```

```
if(FAILED( hr )) {
    Err(_T("Could not create timeline"));
    return hr;
}
```

创建组，并标识为视频组或音频组

组也是一种 DES 对象，通过时间线对象上的接口方法 IAMTimeline::CreateEmptyNode 来创建。然后，我们可以通过给组设置媒体类型，来标识该组是视频组还是音频组。（视频组设置媒体类型为 MEDIATYPE_Video，音频组设置媒体类型为 MEDIATYPE_Audio。）

TimelineTest 例子的实现代码如下：

```
// 创建一个组
hr = pTimeline->CreateEmptyNode( &pVideoGroupObj,
TIMELINE_MAJOR_TYPE_GROUP );
if(FAILED( hr ))
{
    Err(_T("Could not create empty node"));
    return hr;
}
// 标识这个组为视频组
CMediaType VideoGroupType;
VideoGroupType.SetType( &MEDIATYPE_Video );
hr = pVideoGroup->SetMediaType( &VideoGroupType );
if(FAILED( hr ))
{
    Err(_T("Could not set media type"));
    return hr;
}
// 将视频组加入到时间线对象中
hr = pTimeline->AddGroup( pVideoGroupObj );

// ...
// 创建一个组，并标识该组为音频组
hr = pTimeline->CreateEmptyNode( &pAudioGroupObj,
TIMELINE_MAJOR_TYPE_GROUP );
ASSERT( !FAILED( hr ) );

CComQIPtr< IAMTimelineGroup, &IID_IAMTimelineGroup >
pAudioGroup( pAudioGroupObj );
CMediaType AudioGroupType;
AudioGroupType.SetType( &MEDIATYPE_Audio );
hr = pAudioGroup->SetMediaType( &AudioGroupType );
ASSERT( !FAILED( hr ) );
// 将音频组加入到时间线对象中
hr = pTimeline->AddGroup( pAudioGroupObj );
```

在轨道上加入视频过渡处理，并实现过渡子对象的设置

在这之前，我们已经加入了轨道和媒体源。现在我们要在高优先级的轨道上加入过渡处理对象。（TimelineTest 例子中将过渡对象都放在视频组的轨道 1 上。）

TimelineTest 例子的实现代码如下：

```
// 创建过渡对象
CComPtr<IAMTimelineObj> pTrackTransObj;
hr = pTimeline->CreateEmptyNode(&pTrackTransObj,
                                TIMELINE_MAJOR_TYPE_TRANSITION );
// 设置过渡对象的子对象 GUID，以及时间线开始、结束时间
hr = pTrackTransObj->SetSubObjectGUID( CLSID_DxtJpeg ); // SMPTE Wipe
hr |= pTrackTransObj->SetStartStop( TransStart, TransStop );
// 在轨道 1 上获得一个 IAMTimelineTransable 接口
CComQIPtr< IAMTimelineTransable, &IID_IAMTimelineTransable >
pTransable( pTrack2 );
// 将过渡对象加入轨道
hr |= pTransable->TransAdd( pTrackTransObj );
// 设置过渡处理的表现方式（可选），完整效果或是剪影效果
CComQIPtr< IAMTimelineTrans, &IID_IAMTimelineTrans >
pTrackTrans( pTrackTransObj );
#ifdef CUTS_ONLY
    hr = pTrackTrans->SetCutsOnly( TRUE );
#endif  // CUTS_ONLY
```

接着，我们还可以使用 IPropertySetter 接口对象设置过渡对象的属性。比如我们现在用的是 SMPTE Wipe 过渡，需要设置名为 MaskNum 的属性，以指定一种过渡算法，代码如下：

```
CComPtr< IPropertySetter > pTransSetter;
hr = CoCreateInstance( CLSID_PropertySetter, NULL, CLSCTX_INPROC_SERVER,
                       IID_IPropertySetter, (void**) &pTransSetter );
DEXTER_PARAM Param;
CComBSTR ParamName( "MaskNum" ); // the property name
Param.Name = ParamName;
Param.nValues = 1; // how many values we want to set
DEXTER_VALUE Value;
memset( &Value, 0, sizeof( Value ) );
VariantClear( &Value.v );
V_I4( &Value.v ) = 128; // mask number 128
V_VT( &Value.v ) = VT_I4; // integer
hr = pTransSetter->AddProp( Param, &Value );
hr |= pTrackTransObj->SetPropertySetter( pTransSetter );
ASSERT( !FAILED( hr ) );
```

值得注意的是，在图 9.22 中，我们在视频组的轨道 1 上加入了两个过渡处理。因为过渡的默认方向总是从低优先级的轨道转到高优先级的轨道，我们需要为第 2 个过渡改变方向，代码如下：

```
hr |= pTrackTrans->SetSwapInputs( TRUE );
```

在轨道上加入音频效果，并实现动态属性的配置

在 TimelineTest 例子中，创建一个音量调整的音频效果 DES 对象，它的子对象 CLSID 为 CLSID_AudMixer，代码如下：

```
CComPtr< IAMTimelineObj > pTrack3FxObj;
hr = pTimeline->CreateEmptyNode( &pTrack3FxObj,
TIMELINE_MAJOR_TYPE_EFFECT );
ASSERT( !FAILED( hr ) );

// set up effect right
hr = pTrack3FxObj->SetStartStop( TLStart, TLStop );
hr |= pTrack3FxObj->SetSubObjectGUID( CLSID_AudMixer );
ASSERT( !FAILED( hr ) );
```

然后就是加入到音频组的轨道上去，代码如下：

```
CComQIPtr< IAMTimelineEffectable , &IID_IAMTimelineEffectable >
pTrack3Fable( pTrack3 );
hr = pTrack3Fable->EffectInsBefore( pTrack3FxObj, -1 );
ASSERT( !FAILED( hr ) );
```

接下去，我们来看一下如何实现动态属性的设置（我们要让音频的音量在一段时间内慢慢地升高或者降低，表现为一种“淡入淡出”效果），代码如下：

```
CComPtr< IPropertySetter > pVolSetter;
hr = CoCreateInstance( CLSID_PropertySetter, NULL, CLSCTX_INPROC_SERVER,
IID_IPropertySetter, (void**) &pVolSetter );
CComBSTR VolParamName( "Vol" ); // the property name
Param.Name = VolParamName;
Param.nValues = 2; // how many values we want to set
DEXTER_VALUE AudioValue[2];
memset( &AudioValue[0], 0, sizeof( AudioValue[0] ) );
VariantClear( &AudioValue[0].v );
V_R8( &AudioValue[0].v ) = 1.0;     // 开始时候是最高音
V_VT( &AudioValue[0].v ) = VT_R8;
AudioValue[0].rt = 0 * UNITS;     // （相对于效果对象的时间）最高音的时间
memset( &AudioValue[1], 0, sizeof( AudioValue[1] ) );
VariantClear( &AudioValue[1].v );
V_R8( &AudioValue[1].v ) = 0.0;   // 结束时候是静音
V_VT( &AudioValue[1].v ) = VT_R8;
AudioValue[1].rt = 2 * UNITS;     // （相对于效果对象的时间）静音的时间
AudioValue[1].dwInterp = DEXTERF_INTERPOLATE; // 表示音量从最高音到静音之间
                                              // 是慢慢变化的
hr = pVolSetter->AddProp( Param, AudioValue );
hr |= pTrack3FxObj->SetPropertySetter( pVolSetter );
ASSERT( !FAILED( hr ) );
pVolSetter.Release( );
```

从图 9.22 中可以看到，音频组的轨道 0 和轨道 1 在 4~6s 之间都加了一个音频效果：

轨道 0 的音量从最高音慢慢变成静音，轨道 1 的音量从静音慢慢变成最高音。这样，我们可以实现一种模拟的“过渡”效果（虽然 DES 中音频没有像视频那样的过渡处理组件）。

控制引擎的使用

TimelineTest 例子中，控制引擎预览时间线对象总共进行了 3 次。

创建控制引擎的代码如下：

```
hr = CoCreateInstance(
                    CLSID_RenderEngine,
                    NULL,
                    CLSCTX_INPROC_SERVER,
                    IID_IRenderEngine,
                    (void**) &pRenderEngine );
ASSERT( !FAILED( hr ) );
// 将时间线对象设置给控制引擎
hr = pRenderEngine->SetTimelineObject( pTimeline );
ASSERT( !FAILED( hr ) );
// Filter Graph 的前端构建，再连接到 Renderer Filter
hr = pRenderEngine->ConnectFrontEnd( );
hr |= pRenderEngine->RenderOutputPins( );
ASSERT( !FAILED( hr ) );
// 得到 Filter Graph 的各种控制接口
hr = pRenderEngine->GetFilterGraph( &pGraph );
hr |= pGraph->QueryInterface( IID_IMediaEvent, (void**) &pEvent );
hr |= pGraph->QueryInterface( IID_IMediaControl, (void**) &pControl );
hr |= pGraph->QueryInterface( IID_IMediaSeeking, (void**) &pSeeking );
hr |= pGraph->QueryInterface( IID_IVideoWindow, (void**) &pVidWindow );
ASSERT( !FAILED( hr ) );
```

第 1 次预览，代码如下：

```
hr = pControl->Run( );
ASSERT( !FAILED( hr ) );
// 等待所有内容播放完毕
long EventCode = 0;
hr = pEvent->WaitForCompletion( -1, &EventCode );
ASSERT( !FAILED( hr ) );
```

第 2 次预览（重新定位到开始位置，并演示如何只播放在时间线上某段时间内的内容），代码如下：

```
REFERENCE_TIME Start = 0;
#ifdef DO_RENDERRANGE
// 定位到开始位置
hr = pSeeking->SetPositions( &Start, AM_SEEKING_AbsolutePositioning,
NULL, AM_SEEKING_NoPositioning );
ASSERT( !FAILED( hr ) );
// 设置播放的时间段（使用时间线时间限定一个范围）
hr = pRenderEngine->SetRenderRange2( 2.0, 6.0 );
```

```
ASSERT( !FAILED( hr ) );
// 重新调用 ConnectFrontEnd
hr = pRenderEngine->ConnectFrontEnd( );
ASSERT( !FAILED( hr ) );
if(hr == (HRESULT) S_WARN_OUTPUTRESET) {
    hr |= pRenderEngine->RenderOutputPins( );
}
ASSERT( !FAILED( hr ) );
// 开始播放
hr = pControl->Run( );
ASSERT( !FAILED( hr ) );
hr = pEvent->WaitForCompletion( -1, &EventCode );
ASSERT( !FAILED( hr ) );
#endif  // DO_RENDERRANGE
```

第 3 次预览（修改了时间线对象内的编辑信息，然后定位到开始位置重新播放），代码如下：

```
#ifdef DO_RECONNECT
// 对时间线内的过渡对象进行细微的改动
CComPtr< IAMTimelineObj > pTransObj;
REFERENCE_TIME InOut = -1;
pTransable->GetNextTrans( &pTransObj, &InOut );
CComQIPtr< IAMTimelineTrans, &IID_IAMTimelineTrans > pTrans( pTransObj );
pTrans->SetCutsOnly( TRUE );
pTransObj.Release( );
pTrans.Release( );
hr = pTransable->GetNextTrans( &pTransObj, &InOut );
pTrans = pTransObj;
hr |= pTrans->SetCutsOnly( TRUE );
ASSERT( !FAILED( hr ) );
// 定位到开始位置
hr = pSeeking->SetPositions( &Start, AM_SEEKING_AbsolutePositioning,
NULL, AM_SEEKING_NoPositioning );
ASSERT( !FAILED( hr ) );
// 重新调用 ConnectFrontEnd
hr = pRenderEngine->ConnectFrontEnd( );
if(hr == (HRESULT) S_WARN_OUTPUTRESET) {
    hr |= pRenderEngine->RenderOutputPins( );
}
ASSERT( !FAILED( hr ) );
// 开始播放
hr = pControl->Run( );
ASSERT( !FAILED( hr ) );
hr = pEvent->WaitForCompletion( -1, &EventCode );
ASSERT( !FAILED( hr ) );
#endif  // DO_RECONNECT
```

第10章　DVD 播 放

10.1　DVD 基础知识

随着人们物质和精神生活水平的提高，以及多媒体技术的发展，DVD 应用日趋普及。DirectShow 对 DVD 盘片的播放也提供了强大的支持。

首先，来增加一点感性认识，看一下 DVD 光盘的文件系统（采用 MicroUDF 标准）。可以看到，典型情况下，光盘中包含两个文件夹：VIDEO_TS 和 AUDIO_TS（通常是空的，这里不作介绍）。VIDEO_TS 下面一般包含 3 种类型的文件：.VOB 文件、.IFO 文件和.BUP 文件。这些文件都是作什么用的呢？其实，.VOB 文件是用来保存所有 MPEG2 格式的音视频数据的，这些数据包括影片内容、供菜单（Menu）和按钮（Button）以及多种语言字幕用的子图片（Sub-picture）流；.IFO 文件则是控制.VOB 文件播放用的，这个文件中可以找到有关何时以及如何播放.VOB 文件数据的控制信息；而.BUP 文件则是.IFO 文件内容的一个备份（因为.IFO 文件对于保证 DVD 光盘的正确播放起着至关重要的作用）。根据这 3 种文件的特性，也就可以理解它们在光盘介质上的排列顺序了：IFO | VOB | VOB | BUP。

下面再来看一下各个具体的文件。VIDEO_TS.IFO 文件，用于保存 DVD 光盘的视频管理器（Video Manager，简称 VMG）信息，主要是光盘的一些全局信息，比如光盘指定的播放区域、如何显示菜单等。VIDEO_TS.VOB 文件，用于保存显示菜单的数据。类似 VTS_xx_y.VOB 的文件，用于保存各个视频节目。VTS_xx_y.IFO 文件，用于保存对应编号的 VOB 文件的音视频格式信息。

提示　这里的“xx”是节目编号，从 01 到 99，“y”是从 0 到 9 的编号；由于 MicroUDF 文件系统中一个文件最大只能 1 GB，因此大多数影片不得不保存在多个 VOB 文件中。

大致了解了文件系统，再来看一下 DVD 其他方面的基础知识。大家知道，一张 DVD 光盘主要包括 3 种媒体数据：视频（Video）、音频（Audio）和子图片。视频一般采用 MPEG2 压缩算法，视频流支持最多 9 个视角（Angle），支持 Line 21 Closed Caption（模拟电视中的概念，主要是些文字信息）；音频最多支持 8 条不同的流（即 8 种不同语言的配音），支持最多 6 声道的声音格式（具体格式可以是 AC3、MPEG、LPCM、DTS、SDDS 等），支持卡拉 OK；DVD 最多可以支持 32 个子图片流（即提供多种语言的字幕）。DVD 内容最主要的逻辑分类叫做“标题”（Title），一个标题通常代表了一部电影，或者一段视频节目。一张 DVD 光盘最多可以有 99 个标题，每个标题又可以最多分成 999 个“章节”（Chapter）。章节是便于用户随机访问的节点，每播放完一个章节的内容会自动播放下一个章节，或者跳回菜单。

用户与 DVD 的交互，最主要是通过菜单来完成的。菜单有两种：视频管理器菜单

（Video Manager Menu，简称 VMGM，也叫做 Top Menu 或 Title Menu）；视频标题集菜单（Video Title Set Menu，简称 VTSM，也叫做 Root Menu，尽管它不是真正意义上的“根”菜单）。VMGM 允许用户进入主要的标题或者标题集（包含有一组标题）。如果进入标题集，就显示 VTSM 子菜单。VTSM 菜单可以包含进入当前标题集中各个标题的按钮，还有音频、视角、字幕、章节等选择子菜单。VTSM 菜单是可选的，可以不实现；当用户开始播放一张 DVD 光盘时，一般首先看到的就是 VMGM 菜单，除非这张光盘在制作时被设置成自动播放第 1 个标题（此时 DVD 也可以根本不制作 VMGM 菜单）。

DVD 还有父母锁功能。DVD 中的视频内容可以设置父母管理级别（Parental Management Level，PML）；级别可以从 1 到 8，其中 1 是无限制级别，8 是最高限制级别。这样，可以禁止孩子在未经父母同意的情况下观看成人电影。

提示

父母锁功能需要 DVD 播放器的支持。

DVD 还有一个非常重要的抽象概念，就是域（Domain）。通俗地说，也就是 DVD 在特定播放状态下的操作许可。比如“选择按钮”命令必须在菜单状态下才有效，“快进”命令在停止或者菜单状态下是无效的。播放器必须使用域的概念，阻止用户向 DVD 驱动器发送无效的命令。表 10.1 列出了 5 个主要的域以及对应的操作说明。

表 10.1　域及对应的说明

Domain（域）	DVD 导航器的工作内容
First Play（开始播放 DVD 时首先播放的一段视频内容）	读取光盘上的初始部分
Video Manager Menu（显示 VMGM 菜单时）	读取光盘的主菜单，菜单相关的指令都是有效的
Video Title Set Menu（显示 VTSM 菜单时）	读取 VTSM 菜单，或者设置音频、字幕、视角等子菜单，菜单相关的指令都是有效的
Title（播放标题内容时）	菜单相关的指令都是无效的
Stop（停止时）	导航器不再工作，此时可以执行播放指令

另外，DVD 的制作者还可以利用“用户操作控制”（User Operation Controls，UOPs）功能来限制用户的操作。这需要在光盘上记录这些限制记号。比如，大多数光盘都不允许用户在 DVD 播放处于 First Play 域时执行快进或者显示菜单命令。

10.2 应用分析

了解了 DVD 的一些基础知识，接下来，就要介绍 DirectShow 对开发 DVD 应用程序的支持。

10.2.1 支持 MPEG2

由于 DVD 视频内容采用了 MPEG2 的压缩标准，要想播放 DVD 视频，一个 MPEG2 的视频解码 Filter 是必需的。但是很遗憾，DirectShow 并没有提供这个 Filter。

提示 DirectShow 提供了 MPEG-I Stream Splitter、MPEG Video Decoder、MPEG Audio Decoder 3 个 Filter，全面支持 MPEG1 格式。

DirectShow 仅仅提供了两个用于 MPEG2 格式的音视频数据分离的 Filter：MPEG-2 Demultiplexer 和 MPEG-2 Splitter。虽然这两个 Filter 跟 DVD 播放关系不大，但还是来介绍一下它们。

MPEG-2 Demultiplexer

这个 Filter 的 CLSID 为 CLSID_MPEG2Demultiplexer，一般工作在推模式，用于分离 MPEG2 的传输流或者程序流的音频和视频部分。在 Windows XP 下，这个 Filter 也可工作在拉模式下，用于 MPEG2 程序流文件的播放。

提示 MPEG-2 Demultiplexer 工作在推模式时，会暴露一个 IReferenceClock 接口。由于实现了这个接口的 Filter 可以用作 Filter Graph 的参考时钟，此时，MPEG-2 Demultiplexer 就像是一个 Live Source，默认情况下，它就是 Filter Graph 的参考时钟。

MPEG-2 Splitter

这个 Filter 的 CLSID 为 CLSID_MMSPLITTER，仅能工作在拉模式，用于 MPEG2 程序流文件的播放。在 Windows XP 下，这个 Filter 不再被推荐使用（可以使用 MPEG-2 Demultiplexer 来替代它）。

提示 MPEG-2 Demultiplexer 和 MPEG-2 Splitter 都不支持精确到帧的随机定位（Seeking）。

由于 DVD 的.VOB 文件实际上就是 MPEG2 文件，直接播放这些文件时，很可能就要用到上面这两个 Filter 中的其中一个。在 GraphEdit 中播放一个.VOB 文件，构建的 Filter Graph 如图 10.1 所示。

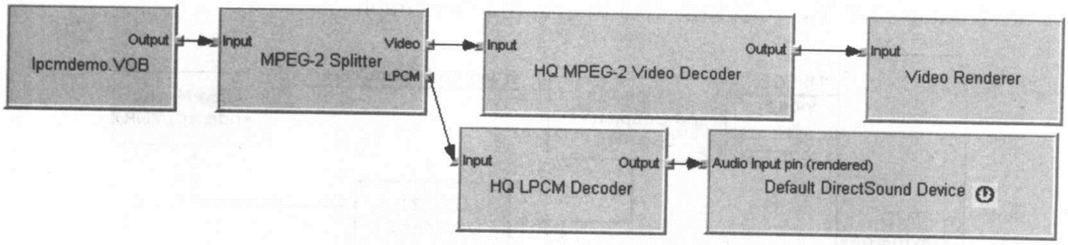

图 10.1 播放.VOB 文件的 Filter Graph

因为一个 MPEG2 文件音频部分的格式不确定，可能是 MPEG2 格式，也可能是 AC-3 格式、LPCM 格式等。为了完整支持 DVD 播放，除了需要准备一个 MPEG2 的视频解码 Filter 外，还需要考虑提供上述各种音频格式的解码 Filter。

10.2.2 DVD 导航器

前面已经介绍了一张 DVD 光盘上的.VOB、.IFO 和.BUP 各个文件的作用，但是当播放一张 DVD 光盘时，怎么来控制和使用它们呢？这是一个很繁琐的任务，光靠前面介绍的那一点 DVD 基础知识是远远不够的。

难道必须去“啃”那本厚厚的《DVD 说明书》吗？有没有其他方法呢？答案是肯定的。DirectShow 提供了一个 DVD 导航器（DVD Navigator），专门来完成这些繁琐的工作。下面，就来介绍这个 Filter。

DVD 导航器的 CLSID 为 CLSID_DVDNavigator，它上面主要实现了两个接口：IDvdInfo2（用于获得光盘的属性和导航状态）和 IDvdControl2（用于设置属性和执行播放操作）。典型的播放 DVD 的 Filter Graph 如图 10.2 所示。

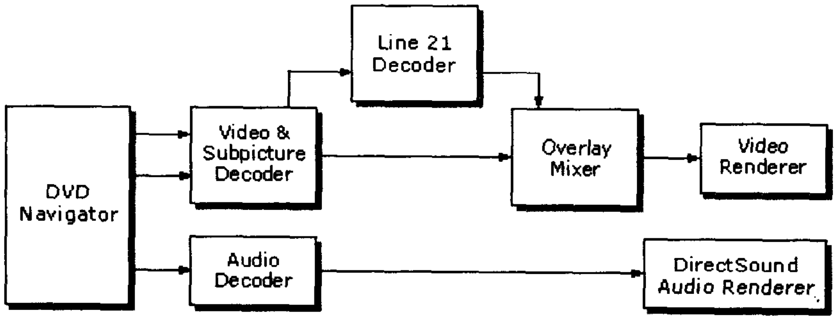

图 10.2　典型的 DVD 播放用的 Filter Graph

从图 10.2 中不难发现，一般一个视频的解码 Filter 要完成视频图像和子图片流两部分解码，然后输出一个合成流。如果 DVD 视频内容中还包含 Line21 CC 信息，还需要使用一个 Line21 Decoder Filter 来解码，然后使用一个 Overlay Mixer 将它与主视频流进行合成。此时，使用一个传统的 Video Renderer。

大家知道，DirectShow 9.0 还发布了一个全新的 Renderer Filter: VMR-9，Windows XP 下还有 VMR-7。于是，构建播放 DVD 的 Filter Graph 时有了更多的选择。

在 Windows XP 下，可以如图 10.3 所示构建 Filter Graph。

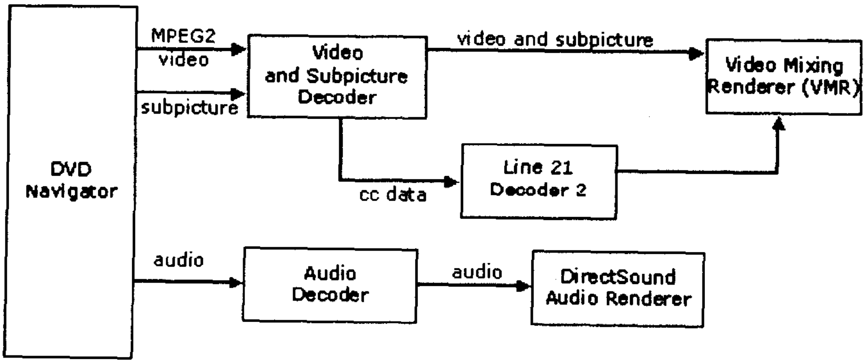

图 10.3　Windows XP 下推荐的播放 DVD 的 Filter Graph

提示

为了兼容 VMR，使用 Line21 Decoder 2 Filter 来解码 CC 信息。这个 Filter 实现的功能其实与图 10.2 中 Line21 Decoder Filter 类似。

当系统中有硬件解码器时（典型的情况，这块硬件集成在显卡上），DirectShow 提供了 Video Port Pin 实现解码后的视频帧绕过主机内存拷贝，直接送入显卡内存显示的一种高效方法。在非 Windows XP 系统下，一般可以构建如图 10.4 所示的 Filter Graph。

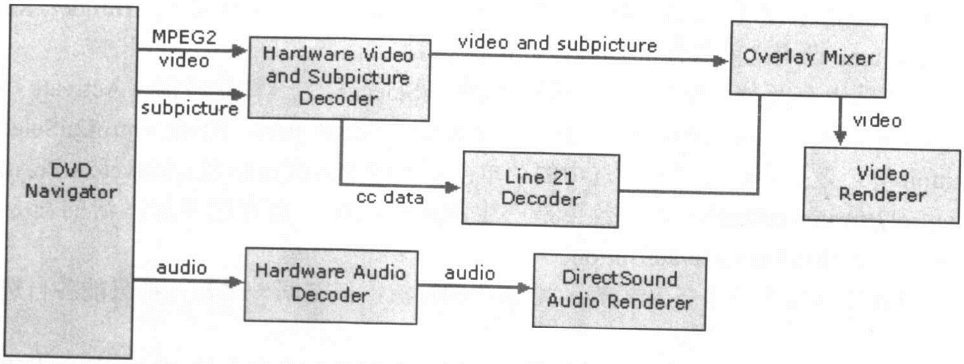

图 10.4 非 Windows XP 下使用硬件解码器播放 DVD

提示

VMR-9 不支持 Video Port Pin 的功能。

而在 Windows XP 下，可以用 Video Port Manager 和 VMR 来替代 Overlay Mixer 和传统的 Video Renderer 的使用，如图 10.5 所示。

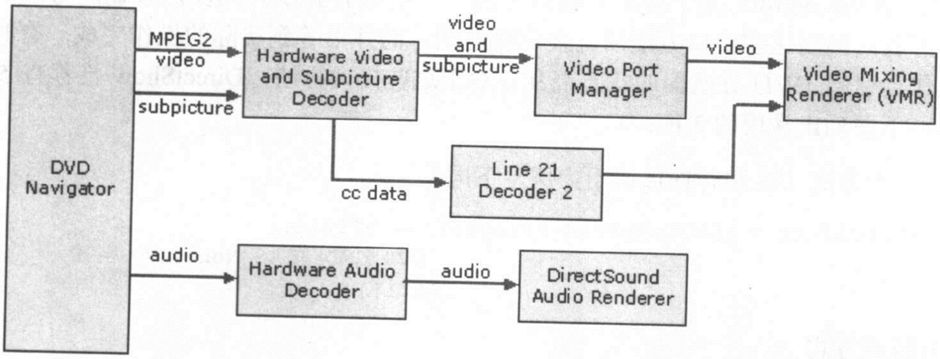

图 10.5 Windows XP 下使用硬件解码器播放 DVD

提示

从上面几个 Filter Graph 参考图可以看出，DVD 导航器总是作为 Source Filter 出现。但是应当记住，DVD 导航器不是 Live Source。

跟构建采集应用的 Filter Graph 类似，DirectShow 也提供了一个辅助构建 DVD 播放的 Filter Graph 的 COM 组件：DVD Graph Builder。可以通过 CoCreateInstance 函数创建这个组件（它的 CLSID 为 CLSID_DvdGraphBuilder，接口为 IDvdGraphBuilder），并很方便地构建用于 DVD 播放的 Filter Graph。其实，开发 DVD 播放的应用程序，最主要的任务还在于 IDvdInfo2 和 IDvdControl2 的各个接口方法的使用。关于这些接口方法的详细说明和用法，还需要参考 DirectShow SDK 文档。下面，仅就编写 DVD 播放程序需要注意的地方，进行一些简单的罗列：

（1）为了得到 DVD 导航器在创建时发出的事件，一般在 IDvdGraphBuilder::RenderDvdVideoVolume 调用之前获得 IMediaEventEx 接口；

（2）实际上有两种菜单命令，一种是选中（Select），一种是激活（Activate）。前者如 IDvdControl2::SelectAtPosition，IDvdControl2::SelectButton，IDvdControl2::SelectRelativeButton 等，效果是高度显示被选中的菜单；后者如 IDvdControl2::ActivateAtPosition，IDvdControl2::ActivateButton 等，效果是产生相应的动作。也有选中加激活的命令，如 IDvdControl2::SelectAndActivateButton。

（3）DVD 最多支持 8 条音频流、32 条子图片流，但在同一时刻，都只能各自选中某一条。

（4）DVD 导航器本身并不强调父母锁功能，而只是把光盘上的 PML 信息发送给应用程序。因此，父母锁功能需要在应用程序上完成。

（5）通过 IDvdInfo2::GetState 调用，可以得到 IDvdState 对象，用以实现“书签”的功能。

（6）注意 DVD 播放时的 Filter Graph 状态，需要同时考虑 DVD 导航器的状态；

（7）支持卡拉 OK 的 DVD，要求音频解码 Filter 实现 AM_KSPROPSETID_DvdKaraoke 属性集（即实现 IKsPropertySet 接口）。

（8）IDvdControl2 关于播放的接口方法，一般都有异步和同步两种调用方式。如果是异步方式，调用这些接口方法后，会立即返回，而并不等到实际的操作完成。有时候，这样操作会引起 DVD 导航器的状态混乱。对于这些接口方法，DirectShow 推荐有 5 种调用方法，下面列出常用的 3 种：

1）异步方式（以 PlayTitle 调用为例，下同）

```
HRESULT hr = pDVDControl2->PlayTitle( uTitle,
                                      DVD_CMD_FLAG_None, // = 0
                                      NULL);
```

2）阻塞方式

```
HRESULT hr = pDVDControl2->PlayTitle( uTitle,
                                      EC_DVD_CMD_FLAG_Block,
                                      NULL);
```

3）使用同步对象

```
IDvdCmd* pObj;
HRESULT hr = pDVDControl2->PlayTitle(uTitle,  0, &pObj);
```

```
if(SUCCEEDED(hr))
{
    pObj->WaitToEnd();
    pObj->Release();
}
```

10.3 实例解剖

DirectShow SDK 提供了一个用于 DVD 播放应用的例子，它位于 SDK 安装目录的 Samples\C++\DirectShow\DVD\DVDSample 下。下面，就来一起学习这个例子。

10.3.1 实现的功能

图 10.6 是 DVDSample 例子的程序界面。这个应用程序完成的主要功能有：

- 同时支持播放 DVD 光盘或硬盘上指定的 DVD 目录；
- 实现 DVD 播放的各种操作；
- 支持全屏播放、单帧步进、书签等功能；
- 实现父母锁功能。

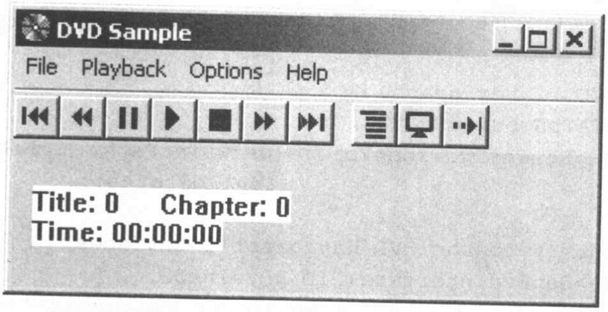

图 10.6 DVDSample 例子的程序界面

10.3.2 实现要点

DVDSample 是一个标准的 Win32 程序，通过这个例子的学习，应掌握如下知识点：

- 使用 DVD Graph Builder 辅助构建 DVD 播放 Filter Graph；
- 控制 DVD 导航器和 Filter Graph 的状态；
- 处理视频窗口问题；
- 实现单帧步进功能；
- 实现快进、快退；
- 实现书签功能；
- 选择字幕、配音、视角、标题、章节，以及指定时间播放；
- 获得 DVD 音视频属性。

使用 DVD Graph Builder 辅助构建 DVD 播放 Filter Graph

DVDSample 例子设计了一个类 CDvdCore，封装了构建 DVD 播放 Filter Graph 的所有操作。下面使用 DVD Graph Builder 组件来辅助 Filter Graph 的构建，大致过程如下：

```
// Create an instance of the DVD Graph Builder object.
HRESULT hr;
hr = CoCreateInstance(CLSID_DvdGraphBuilder,
                      NULL,
                      CLSCTX_INPROC_SERVER,
                      IID_IDvdGraphBuilder,
                      reinterpret_cast<void**>(&m_pIDvdGB));
   ...
// Get a pointer to the filter graph manager.
hr = m_pDvdGB->GetFiltergraph(&m_pGraph);

    // It is important to QI for IMediaEventEx *before* building the graph
    // so that the app can catch all of the DVD Navigator's initialization
    // events.Once an app QI's for IMediaEventEx, DirectShow will start
    // queuing up events and the app will receive them when it set the
    // notify window (in InitMessageSink for this sample). If the app
    // does not QI for IMediaEventEx before building the graph, these
    // events will just be lost.
hr = m_pGraph->QueryInterface(IID_IMediaEventEx,
                      reinterpret_cast<void**>(&m_pIME)) ;

// Build the DVD filter graph.
AM_DVD_RENDERSTATUS buildStatus;
hr = m_pIDvdGB->RenderDvdVideoVolume(pszwDiscPath, m_dwRenderFlags,
                      &buildStatus);

// Get the pointers to the DVD Navigator interfaces.
hr = m_pIDvdGB->GetDvdInterface(IID_IDvdInfo2,
                      reinterpret_cast<void**>(&m_pIDvdI2));
hr = m_pIDvdGB->GetDvdInterface(IID_IDvdControl2,
                      reinterpret_cast<void**>(&m_pIDvdC2));
...
// Use the graph pointer to get a pointer to IMediaControl,
// for controlling the filter graph as a whole.
hr = m_pGraph->QueryInterface(IID_IMediaControl,
                      reinterpret_cast<void**>(&m_pIMC));
...
// Get a pointer to IMediaEventEx,
// used for handling DVD and other filter graph events.
hr = m_pGraph->QueryInterface(IID_IMediaEventEx,
                      reinterpret_cast<void**>(&m_pME));
...
// Use the graph builder pointer again to get the IVideoWindow interface,
// to set the window style and message-handling behavior of the video
//renderer filter.
hr = m_pIDvdGB->GetDvdInterface(IID_IVideoWindow,
                      reinterpret_cast<void**>(&m_pIVW));
```

```
hr = m_pDvdGB->GetDvdInterface(IID_IAMLine21Decoder,
                               reinterpret_cast<void**>(&pL21Dec));
```

具体的实现可以参见 CDvdCore::BuildGraph()。

控制 DVD 导航器和 Filter Graph 的状态

DVDSample 例子定义了如下的一个状态机：

```
// DirectShow Graph state
enum GRAPH_STATE
{
    Uninitialized = 0,
    Graph_Stopped1, // stopped without reset
    Graph_Stopped2, // stopped with reset
    Nav_Stopped,
    Playing,
    Graph_Paused,
    Nav_Paused,
    Scanning
} ;
```

可以看出，CDvdCore 类中有一个 GRAPH_STATE 类型成员变量：

```
GRAPH_STATE m_eState;
```

还有相关的内联函数：

```
inline GRAPH_STATE GetState(void) { return m_eState; }
inline void SetState(GRAPH_STATE state) { m_eState = state; };
```

m_eState 在不同的情况下的取值如下：

- 初始值为 Uninitialized；
- 当 CDvdCore::BuildGraph 调用成功后，就置为 Graph_Stopped2；
- 当开始正常播放 DVD，即调用 CDvdCore::Play 时，置为 Playing；
- 当要停止 DVD 的播放，即调用 CDvdCore::Stop 时，如果是第一次按下停止按钮，则置为 Graph_Stopped1，如果是第二次按下停止按钮，则置为 Graph_Stopped2；
- 当执行快退命令，即调用 CDvdCore::Rewind 时，则置为 Scanning；
- 当要暂停 DVD 播放，即调用 CDvdCore::Pause 时，则置为 Nav_Paused。当实现单帧步进时，调用 CDvdCore::FrameStep()，m_eState 也置为 Nav_Paused；
- 当释放 DVD 播放所使用的资源时，置为 Uninitialized（参见 CDvdCore::ReleaseInterfaces）。

处理视频窗口问题

DVDSample 例子创建了一个主窗口之外的 WS_OVERLAPPEDWINDOW 风格的视频窗口，用于播放 DVD 视频、接收键盘鼠标输入、处理 DVD 事件等。下面来看 CDvdCore::InitMessageSink 的实现：

```
// 自定义消息 WM_DVD_EVENT 标识 Filter Graph Manager 发出的事件
```

```
HRESULT hr;
hr = m_pIME->SetNotifyWindow(reinterpret_cast<OAHWND>(m_hWnd),
                             WM_DVD_EVENT, 0);
ASSERT(SUCCEEDED(hr));
// 将 Video Renderer 的视频窗口设置为应用程序视频窗口的子窗口
long lStyle;
m_pIVW->get_WindowStyle(&lStyle);
m_pIVW->put_WindowStyle( (lStyle | WS_CHILD) & ~(WS_CAPTION | WS_BORDER
                        | WS_THICKFRAME) );
m_pIVW->put_Owner(reinterpret_cast<OAHWND>(m_hWnd));
RECT rect;
GetClientRect(m_hWnd, &rect);
m_pIVW->SetWindowPosition(0, 0, rect.right, rect.bottom);
// 让视频窗口可以接收键盘、鼠标输入
hr = m_pIVW->put_MessageDrain(reinterpret_cast<OAHWND>(m_hWnd)) ;
ASSERT(SUCCEEDED(hr)) ;
```

下面再来看视频窗口的消息处理函数 CDvdCore::WndProc:

```
switch (uMessage)
{
    // 定时器：当在全屏播放模式，鼠标静止一段时间后自动隐藏
    case WM_TIMER:
        return pCore->OnMouseTimer(wParam, lParam);
    case WM_SIZE:        // 处理窗口改变消息
        return pCore->OnSize(wParam, lParam);
    case WM_KEYUP:       // 处理键盘输出，参见 CDvdCore::OnKeyEvent
        return pCore->OnKeyEvent(wParam, lParam);
    case WM_DVD_EVENT:   // 处理 DVD 事件，参见 CDvdCore::OnDvdEvent
        return pCore->OnDvdEvent(uMessage, wParam, lParam);
    case WM_MOUSEMOVE:   // 处理鼠标输入，参见 CDvdCore::OnMouseEvent
    case WM_LBUTTONUP:
    case WM_RBUTTONUP:
    case WM_MBUTTONUP:
        return pCore->OnMouseEvent(uMessage, wParam, lParam);
    case WM_CLOSE:       // 当视频窗口关闭时，同时退出应用程序
        return pCore->OnClose();
}
```

实现单帧步进功能

单帧步进主要靠 Filter Graph Manager 上的 IVideoFrameStep 接口来实现。下面来看 DVDSample 例子的 CDvdCore::FrameStep 函数实现:

```
// Get the Frame Stepping Interface
IVideoFrameStep* pFS;
HRESULT hr;
hr = m_pGraph->QueryInterface(__uuidof(IVideoFrameStep), (PVOID *)&pFS);
// check if this decoder can step
hr = pFS->CanStep(0L, NULL);
if (S_FALSE == hr)
{
```

```
        DbgLog((LOG_ERROR, 2, TEXT("WARNING: Decoder can't step"),
                szDirectory, hr)) ;
        pFS->Release();
        return false;
    }
    // 首先必须将 Filter Graph 置于暂停状态
    m_pIMC->Pause();
    SetState(Graph_Paused);
    // 前进一帧
    hr = pFS->Step(1, NULL);
    bool retVal = SUCCEEDED(hr) ? true : false;
    // clean up after ourselves
    pFS->Release();
```

实现快进、快退

快进、快退主要通过 IDvdControl2::PlayForwards 和 IDvdControl2::PlayBackwards 来实现。对于 DVDSample 例子，可以参考 CDvdCore::FastForward 和 CDvdCore::Rewind 的函数实现。

实现书签功能

DVDSample 例子中，保存一个书签（到硬盘文件）参见 CDvdCore::SaveBookmark 函数的实现：

```
// 首先得到 DVD 状态
IDvdState * pBookmark;
hr = m_pIDvdI2->GetState (&pBookmark);
IStream * pStream;
// 通过 StgOpenStorage 或 StgCreateDocfile 打开一个标签文件，并得到 IStream 接口
// ...
// 然后将 pBookmark 这个状态通过 IStream 接口保存到硬盘文件
IPersistStream * pPersistStream;
hr = pBookmark->QueryInterface(IID_IPersistStream,
reinterpret_cast<void**>(&pPersistStream));
if (SUCCEEDED(hr))
{
    hr = OleSaveToStream(pPersistStream, pStream);
    pPersistStream->Release();
    pStorage->Release();
    pStream->Release();
    pBookmark->Release();
    if (SUCCEEDED(hr))
        return true;
    else
        return false;
}
```

恢复书签参见 CDvdCore::RestoreBookmark 函数的实现：

```
// 打开书签文件
IStorage * pStorage;
HRESULT hr = StgOpenStorage(g_szBookmarkFilename, NULL, STGM_READ |
                            STGM_SHARE_EXCLUSIVE,NULL, 0, &pStorage);
// open a stream object within that file
IStream * pStream;
hr = pStorage->OpenStream(g_szBookmarkName, NULL, STGM_READ |
                          STGM_SHARE_EXCLUSIVE, 0,&pStream);
// load the bookmark in from the stream
IDvdState * pBookmark;
hr = OleLoadFromStream(pStream, IID_IDvdState,
                       reinterpret_cast<void**>(&pBookmark));
pStream->Release();
pStorage->Release();
// 将书签文件中保存的状态设置到 DVD 导航器中
hr = m_pIDvdC2->SetState(pBookmark, DVD_CMD_FLAG_Block, NULL);
pBookmark->Release();
```

选择字幕、配音、视角、标题、章节，以及指定时间播放

DVDSample 例子的 Dialogs.h 中定义了一系列的包装对话框，要完成上述功能，可以参考 Dialogs.cpp 中这些对话框的具体实现。

获得 DVD 音视频属性

参见 CDvdCore::GetSPAttributes，来获取当前字幕属性，比如字幕语言等；

参见 CDvdCore::GetVideoAttributes，来获取当前 DVD 视频属性，包括视频的压缩格式、图像分辨率等；

参见 CDvdCore::GetAudioAttributes，来获取当前 DVD 音频属性，包括音频的压缩格式、声道数、采样频率、量化精度等。

第 11 章　Windows Media 应用

11.1　应 用 分 析

大家知道，微软公司定义了自己的一种媒体文件类型，叫做 ASF（Advanced Systems Format）。其实，ASF 只是一种文件“容器”，它本身并没有规定音视频的压缩格式；在 ASF 文件中，可以包含任何格式的压缩的（包括 MPEG-4）或非压缩的数据。

提示　微软公司提供了独立于 DirectX 开发环境以外的 Windows Media Format SDK（以下简称 WMFS），专门用于支持客户开发 ASF 相关的应用。

那么，能不能在 DirectShow 中直接开发 ASF 的应用呢？答案是肯定的。在 DirectShow 中，有一系列的 Filter 用于 ASF 文件的播放和生成，只是这些 Filter 的版本关系比较复杂，必须先来整理一下。

最早从 Windows Media Player（微软公司的媒体播放器）6.4，DirectShow 中就有了一个名为 Windows Media Source 的 Filter，来支持 ASF 文件的播放。这个 Source Filter 默认与 .asf、.wmv、.wma 扩展名文件关联（即在调用 IGraphBuilder::RenderFile 或 IGraphBuilder::AddSourceFilter 来播放上述扩展名的文件时，Filter Graph Manager 会自动使用 Windows Media Source Filter）。图 11.1 是典型的使用 Windows Media Source Filter 来播放 ASF 文件的 Filter Graph 参考图。

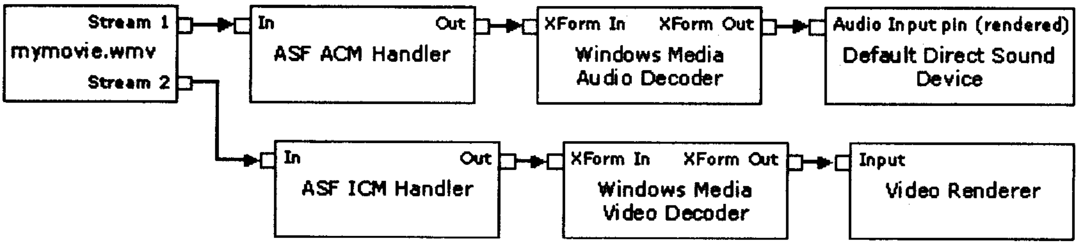

图 11.1　使用 Windows Media Source Filter 播放 ASF 文件

从 DirectShow 8.0 开始，出现了另外两个 Filter：WM ASF Reader 和 WM ASF Writer。并且随着 DirectShow 版本的更新，这两个 Filter 的版本也在不断更新（DirectShow 8.1 中的这两个 Filter 使用了 WMFS 7.0，DirectShow 9.0 中的使用了 WMFS 7.1）。但是，尽管如此，为了保持向后兼容性，DirectShow 还是让 Windows Media Source Filter 作为播放.asf、.wmv、.wma 文件的默认 Source Filter。直到 WMFS 9.0（包含有最新版本的 WM ASF Reader 和 WM ASF Writer）的发布，WM ASF Reader 才真正完全取代了 Windows Media Source Filter。图 11.2 是典型的使用 WM ASF Reader 来播放 ASF 文件的

Filter Graph 参考图。

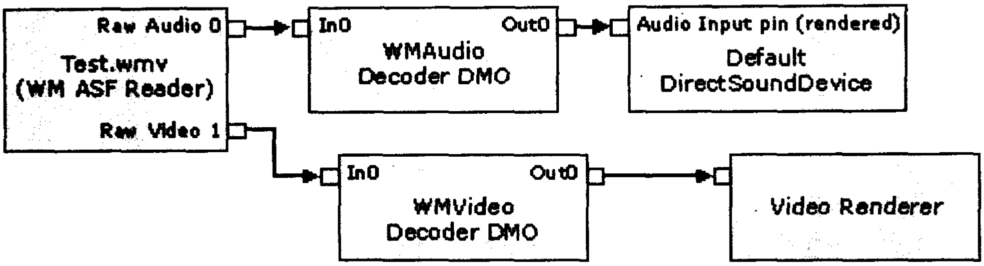

图 11.2　使用 WM ASF Reader 播放 ASF 文件

理清了上述的版本关系，编写一个播放一般的 ASF 文件的应用程序，应该没有什么问题了。而播放另外一些特殊的 ASF 文件，比如经过了 DRM （Digital Rights Management）保护的文件，我们还需要提供一些特殊的处理（比如进行许可认证）。

下面，再来看一下如何生成 ASF 文件。我们已经知道，ASF 是一个文件“容器”。.asf 扩展名是这种格式文件的“统称”，而如果生成的 Window Media 文件只包含音频，一般使用.wma 扩展名，如果只包含视频、或同时包含音视频，一般使用.wmv 扩展名。使用 WM ASF Writer，有一个重要的概念就是配置（Profile）。通过配置，可以设定 WM ASF Writer 有几个输入 Pin、生成文件的比特率、压缩质量等等。

提示　使用 SDK 安装目录下的 Bin\DXUtils\ProfileEnum.exe 工具，可以枚举得到系统中所有的 ASF 配置。

当 WM ASF Writer 创建时，该 Filter 使用了一个系统默认的配置。如果这个配置不符合具体应用，可以通过 Filter 上的 IConfigAsfWriter 接口改变配置。值得注意的是，在配置 WM ASF Writer 之前，必须先把它加入到 Filter Graph 中，而必须等到配置完成后才能进行 Pin 的连接；运行 Filter Graph 之前，WM ASF Writer 的所有输入 Pin 都要求完成连接，而且各个输入 Pin 上的输入数据要求都是打好时间戳的。还有一个重要的概念需要提醒，WMFS 规定，ASF 文件必须以音频为基准来播放。所以，一般 WM ASF Writer 总是需要一个音频的输入 Pin，即使这个 Pin 上只是输入一些静音的、低码率的数据。

图 11.3 是常见的 WM ASF Writer 3 种应用，可供参考。

构建上述 Filter Graph，一般有两种切实可行的方法：一种是，一个一个地加入各个 Filter，配置 WM ASF Writer（如果系统默认配置不符合具体应用时），然后完成整个 Filter 链路的连接；另一种方法是，使用 IGraphBuilder::RenderFile 首先构建源文件的播放链路，然后将 Renderer Filter 从 Filter Graph 中移除，再加入 WM ASF Writer，进行必要的配置后完成最终的连接。

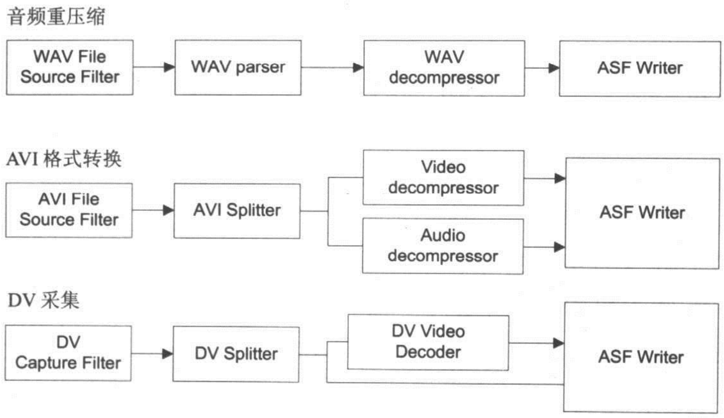

图 11.3 WM ASF Writer 三种应用

11.2 实例解剖

DirectShow SDK 提供了一个用于 ASF 应用的例子，它位于 SDK 安装目录的 Samples\C++\DirectShow\Misc\ASFCopy 下。下面，就来一起学习这个例子。

提示 编译这个例子，需要首先安装 Windows Media Format SDK，并且配置好 VC 的 Include 和 Library 系统路径。

11.2.1 实现的功能

图 11.4 是 ASFCopy 例子的程序界面。这个应用程序可以将一个指定的媒体文件转化为 ASF 文件。典型的命令行执行如下：ASFCopy.exe /v /p 8 F:\aviDemo.AVI F:\asfDemo.asf，即使用系统 8 号配置将源文件 F:\aviDemo.AVI 转化为 ASF 格式的文件 F:\asfDemo.asf。

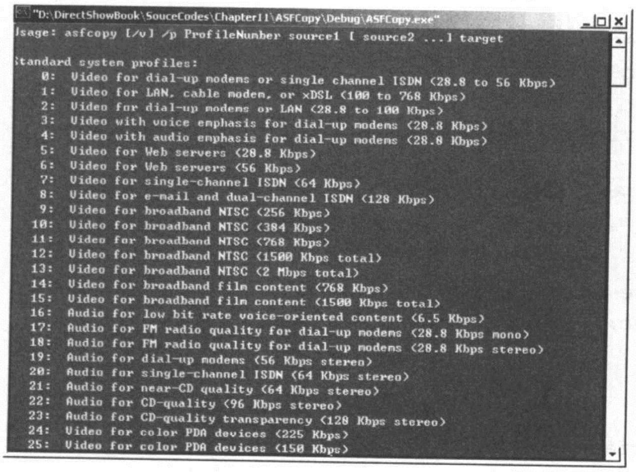

图 11.4 ASFCopy 例子的程序界面

11.2.2 实现要点

ASFCopy 是一个控制台程序，通过这个例子的学习，将掌握如下知识点：

- 为 ASF 文件提供认证服务；
- 枚举系统中的 ASF 配置；
- 为 WM ASF Writer 提供一个配置；
- 提高生成 ASF 文件的速度；
- WaitForCompletion 的实现参考。

为 ASF 文件提供认证服务

主要是一个带有 IServiceProvider 接口的 COM 组件的实现：

```
class CKeyProvider : public IServiceProvider
{
    public:
        //
        // IUnknown interface
        //
        STDMETHODIMP QueryInterface(REFIID riid, void ** ppv);
        STDMETHODIMP_(ULONG) AddRef();
        STDMETHODIMP_(ULONG) Release();
        CKeyProvider();
        // IServiceProvider
        STDMETHODIMP QueryService(REFIID siid, REFIID riid, void **ppv);
    private:
        volatile LONG m_cRef;
};
CKeyProvider::CKeyProvider() : m_cRef(0)
{
}
ULONG CKeyProvider::AddRef()
{
    InterlockedIncrement( &m_cRef );
    return m_cRef;
}
ULONG CKeyProvider::Release()
{
    ASSERT(m_cRef > 0);
    ULONG urc = (ULONG) InterlockedDecrement( &m_cRef );
    if (urc == 0 )
        delete this;
    return urc;
}

//
// QueryInterface
//
// We only support IUnknown and IServiceProvider
```

```
//
HRESULT CKeyProvider::QueryInterface(REFIID riid, void ** ppv)
{
    if(riid == IID_IServiceProvider || riid == IID_IUnknown)
    {
        *ppv = (void *) static_cast<IServiceProvider *>(this);
        AddRef();
        return NOERROR;
    }
    return E_NOINTERFACE;
}
STDMETHODIMP CKeyProvider::QueryService(REFIID siid, REFIID riid, void
                                        **ppv)
{
    if(siid == __uuidof(IWMReader) && riid == IID_IUnknown)
    {
        IUnknown *punkCert;

        // 创建证书
        HRESULT hr = WMCreateCertificate(&punkCert);
        if(SUCCEEDED(hr))
            *ppv = (void *) punkCert;
        else
        {
            printf("CKeyProvider::QueryService failed to create
                    certificate!  hr=0x%x\n", hr);
            *ppv = (void *) NULL;
        }
        return hr;
    }
    return E_NOINTERFACE;
}
```

组件实现好后，使用方法如下：

```
CKeyProvider prov;
prov.AddRef();  // Don't let COM try to free our static object
// Give the graph a pointer to us for callbacks & QueryService
hr = pGraph->QueryInterface(IID_IObjectWithSite,
                            (void**)&pObjectWithSite);
if(SUCCEEDED(hr))
{
    hr = pObjectWithSite->SetSite((IUnknown *) (IServiceProvider *)
                                  &prov);
    if(FAILED(hr))
    {
        printf("Failed to set service provider!  hr=0x%x\n", hr);
        return hr;
    }
}
```

枚举系统中的 ASF 配置

主要是使用 IWMProfileManager 和 IWMProfileManager2 接口，可以枚举不同版本的 ASF 配置。版本定义如下：

```
typedef
enum WMT_VERSION
    {       WMT_VER_4_0    = 0x40000,
            WMT_VER_7_0    = 0x70000,
            WMT_VER_8_0    = 0x80000,
            WMT_VER_9_0    = 0x90000
    }       WMT_VERSION;
```

ASFCopy 例子枚举了 WMT_VER_7_0 版本的所有配置，具体实现参见 ListProfiles 函数。

为 WM ASF Writer 提供一个配置

根据外部指定的一个配置号，首先要返回一个 IWMProfile 接口对象（仍然是使用 IWMProfileManager 和 IWMProfileManager2 接口）。参见 ASFCopy 例子的 MapProfileId-ToProfile 函数实现。

然后要将这个配置设给 WM ASF Writer（使用 Filter 上的 IConfigAsfWriter 接口），整个过程参考如下：

```
    hr = pMux->QueryInterface(IID_IConfigAsfWriter, (void **)
&pConfigAsfWriter);
    if(SUCCEEDED(hr))
    {
        if (fVerbose)
            printf("Setting profile to %d\r\n", dwProfile);
        CComPtr<IWMProfile> pProfile;
        hr = MapProfileIdToProfile(dwProfile, &pProfile);
        if(FAILED(hr)) {
            printf("Failed to map profile ID!  hr=0x%x\n", hr);
            return hr;
        }
        // Note that the ASF writer will not run if the number of streams
        // does not match the profile.
        hr = pConfigAsfWriter->ConfigureFilterUsingProfile(pProfile);
        if(FAILED(hr)) {
          printf("Failed to configure filter to use profile!  hr=0x%x\n",
                 hr);
          return hr;
        }
    }
```

提高生成 ASF 文件的速度

因为并不需要预览媒体文件，所以可以让 Filter Graph 不使用参考时钟，以使数据流中的 Sample 以最快的速度传送。ASFCopy 例子的实现如下：

```
HRESULT SetNoClock(IFilterGraph *pGraph)
{
    if (!pGraph)
        return E_POINTER;
    // Keep a useless clock from being instantiated....
    IMediaFilter *pFilter=NULL;
    HRESULT hr = pGraph->QueryInterface(IID_IMediaFilter, (void **)
                                        &pFilter);
    if(SUCCEEDED(hr))
    {
        hr = pFilter->SetSyncSource(NULL);
        if (FAILED(hr))
            printf("SetNoClock: Failed to set sync source!  hr=0x%x\n",
                   hr);
        pFilter->Release();
    }
    else
    {
        printf("SetNoClock: Failed to QI for media filter!  hr=0x%x\n",
               hr);
    }
    return hr;
}
```

WaitForCompletion 的实现参考

ASFCopy 例子中，在等待整个 Filter Graph 运行结束时，并没有直接调用 IMediaEvent::WaitForCompletion(INFINITE, &lEvCode)，而是用一个 While 循环，之间加入了消息循环。实现参考如下：

```
void WaitForCompletion( IGraphBuilder *pGraph )
{
    HRESULT hr;
    LONG lEvCode = 0;
    IMediaEvent *pEvent;
    if (!pGraph)
        return;
    hr = pGraph->QueryInterface(IID_IMediaEvent, (void **) &pEvent);
    if (SUCCEEDED(hr))
    {
        printf("Waiting for completion...\n This could take several
               minutes, "
               "depending on file size and selected profile.\n");
        do
        {
            MSG Message;
            while(PeekMessage(&Message, NULL, 0, 0, TRUE))
            {
                TranslateMessage(&Message);
                DispatchMessage(&Message);
            }
```

```
            hr = pEvent->WaitForCompletion(10, &lEvCode);
        } while(lEvCode == 0);
        pEvent->Release();
    }
    else
        printf("QI failed for IMediaEvent interface!\n");
}
```

第 3 部分　SDK 典型源码分析

第 12 章　Source Filter 例子

12.1　拉模式例子

这一节，将介绍一个拉模式的 Source Filter 的实现，它位于 SDK 安装目录 Samples\C++\DirectShow\Filters\Async 下。

12.1.1　实现的功能

用 VC 打开 async.dsw 文件，可以看到如下 3 个项目：

- asynbase：是一个静态库项目，实现了异步读写的基类；
- async：使用了 asynbase 库，实现了一个拉模式的 Source Filter；
- memfile：使用了 asynbase 库，在应用程序中实现了一个不注册的拉模式 Source Filter，然后进行媒体文件的播放。

提示　由于 async 和 memfile 都是使用 asynbase 库的例子，应用原理类似。为了简单起见，下面只重点介绍 asynbase 和 async 两个项目的实现。

我们知道，DirectShow 提供了一个标准的 File Source (Async.)（它的 CLSID 为 CLSID_AsyncReader），作为一般媒体文件播放的 Source Filter。实际上，async 实现了类似的功能；可以把它看成是一个简化了的 File Source (Async.)。

编译 async 项目，完成后再将 Filter 文件注册。这样，可以用 GraphEdit 找到一个名为 Sample File Source (Async.)的 Filter；再将这个 Filter 加入到 Filter Graph 中，在弹出的一个对话框中，选择一个媒体文件（比如选择一个名为 Demo1.mpg 的 MPEG1 文件）。在 Source Filter 的输出 Pin 上右击，执行菜单项“Render Pin”，构建如图 12.1 所示完整的 Filter Graph。

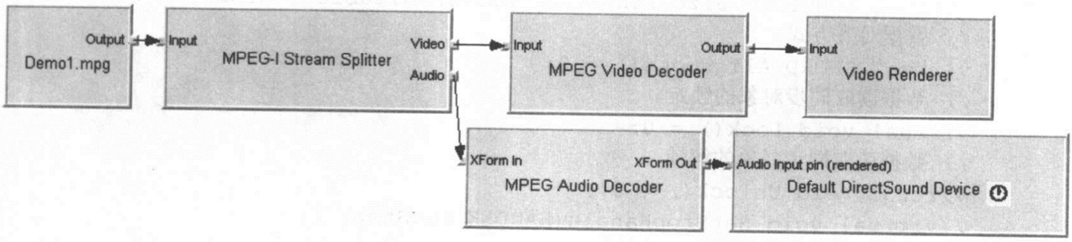

图 12.1　使用 Sample File Source (Async.)构建的 Filter Graph

如图 12.1 所示，拉模式的 Source Filter 后面总是接一个带有“拉”数据能力的输入 Pin 的 Filter（这种 Filter 一般叫做 Splitter 或者 Parser 等）。运行这个 Filter Graph，就可以

看到 Demo1.mpg 文件中的内容演示。

> **提示** Sample File Source (Async.)不支持 AVI 文件，因为这个 Filter 的输出 Pin 上使用(MEDIATYPE_Stream, MEDIASUBTYPE_NULL)的媒体类型，而这种媒体类型不能被 AVI Splitter 的输入 Pin 所接受。

12.1.2　实现要点

首先来学习 asynbase 项目的实现。这个项目中实现了以下几个类：CAsyncStream、CAsyncRequest、CAsyncIo、CAsyncOutputPin、CAsyncReader 等。CAsyncOutputPin 和 CAsyncReader 类的继承关系如图 12.2 所示。

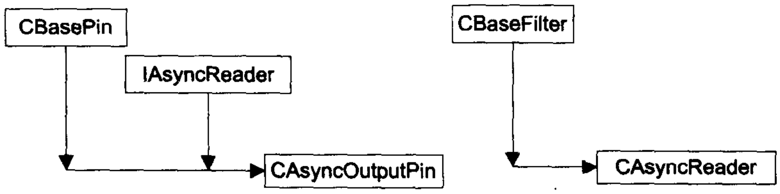

图 12.2　asynbase 项目的类继承关系

CAsyncStream 是一个纯虚类（它的成员函数都是纯虚函数），用以标识一个数据流“源”：

```
class CAsyncStream
{
public:
    virtual ~CAsyncStream() {};
    // 设置流数据的读取位置
    virtual HRESULT SetPointer(LONGLONG llPos) = 0;
    // 外部调用该函数读取流中指定数量的数据
    virtual HRESULT Read(PBYTE pbBuffer,
                         DWORD dwBytesToRead,
                         BOOL bAlign,
                         LPDWORD pdwBytesRead) = 0;
    // 获得数据流的总长度和当前已经可以读取的长度
    virtual LONGLONG Size(LONGLONG *pSizeAvailable = NULL) = 0;
    // 数据对齐方式
    virtual DWORD Alignment() = 0;
    // 数据读取同步对象的锁定
    virtual void Lock() = 0;
    // 数据读取同步对象的解锁
    virtual void Unlock() = 0;
    //virtual void SetStopHandle(HANDLE hevStop) {}
};
```

CAsyncRequest 类表示一个输入和输出的请求，它的成员函数 Complete 实际地执行这个请求。

CAsyncIo 类实现了数据输入和输出的控制，包括同步模式和异步模式。这个类内部

维持了两个列表：m_listWork 和 m_listDone，列表元素为 CAsyncRequest 类型的对象指针。m_listWork 存放等待数据读取的所有请求，m_listDone 存放已经完成的请求。当外部要求异步数据读取时，一般调用 CAsyncIo::Request 生成一个请求，然后放到 m_listWork 列表中等待执行；此时外部需要调用 CAsyncIo::WaitForNext 等待一个请求执行的完成。下面是这两个函数的实现：

```
HRESULT CAsyncIo::Request(
                LONGLONG llPos,
                LONG lLength,
                BOOL bAligned,
                BYTE * pBuffer,
                LPVOID pContext,
                DWORD dwUser)
{
    if(bAligned)
    {
        if(!IsAligned(llPos) ||
            !IsAligned(lLength) ||
            !IsAligned((LONG) pBuffer))
        {
            return VFW_E_BADALIGN;
        }
    }
    // 生成一个新的请求
    CAsyncRequest* pRequest = new CAsyncRequest;
    if (!pRequest)
        return E_OUTOFMEMORY;
    // 配置请求参数
    HRESULT hr = pRequest->Request(this,
                                   m_pStream,
                                   llPos,
                                   lLength,
                                   bAligned,
                                   pBuffer,
                                   pContext,
                                   dwUser);
    if(SUCCEEDED(hr))
    {
        // might fail if flushing
        // 将请求放入 m_listWork 列表中
        hr = PutWorkItem(pRequest);
    }
    if(FAILED(hr))
    {
        delete pRequest;
    }
    return hr;
}
HRESULT CAsyncIo::WaitForNext(
```

```
    DWORD dwTimeout,
    LPVOID * ppContext,
    DWORD  * pdwUser,
    LONG   * pcbActual)
{
    CheckPointer(ppContext,E_POINTER);
    CheckPointer(pdwUser,E_POINTER);
    CheckPointer(pcbActual,E_POINTER);
    // some errors find a sample, others don't. Ensure that
    // *ppContext is NULL if no sample found
    *ppContext = NULL;
    // wait until the event is set, but since we are not
    // holding the critsec when waiting, we may need to re-wait
    for(;;)
    {
        // 当请求执行完成后，m_evDone 事件会置为有效状态
        if(!m_evDone.Wait(dwTimeout))
        {
            // timeout occurred
            return VFW_E_TIMEOUT;
        }
        // 从 m_listDone 列表中取出一个已经完成的请求
        CAsyncRequest* pRequest = GetDoneItem();
        if(pRequest)
        {
            // found a completed request
            // check if ok
            HRESULT hr = pRequest->GetHResult();
            if(hr == S_FALSE)
            {
                // this means the actual length was less than
                // requested - may be ok if he aligned the end of file
                if((pRequest->GetActualLength() +
                    pRequest->GetStart()) == Size())
                {
                    hr = S_OK;
                }
                else
                {
                    // it was an actual read error
                    hr = E_FAIL;
                }
            }
            // return actual bytes read
            *pcbActual = pRequest->GetActualLength();
            // return his context
            *ppContext = pRequest->GetContext();
            *pdwUser = pRequest->GetUser();
      // 记得这里将请求对象删除！
           delete pRequest;
```

```
            return hr;
        }
        else
        {
            //  Hold the critical section while checking the list state
            CAutoLock lck(&m_csLists);
            if(m_bFlushing && !m_bWaiting)
            {
                // can't block as we are between BeginFlush and EndFlush
                // but note that if m_bWaiting is set, then there are
                // some items not yet complete that we should block for.
                return VFW_E_WRONG_STATE;
            }
        }
        // done item was grabbed between completion and
        // us locking m_csLists.
    }
}
```

那么在异步模式下，实际的请求是在哪里执行的呢？可以发现，CAsyncIo 类的实现中还使用了一个工作者线程，线程处理函数如下：

```
DWORD CAsyncIo::ThreadProc(void)
{
    HANDLE ahev[] = {m_evStop, m_evWork};
    for(;;)
    {
        DWORD dw = WaitForMultipleObjects(2,
                                          ahev,
                                          FALSE,
                                          INFINITE);
        if(dw == WAIT_OBJECT_0+1)
        {
            // requests need processing
            ProcessRequests();
        }
        else
        {
            // any error or stop event - we should exit
            return 0;
        }
    }
}
```

下面再来看 ProcessRequests 函数的实现：

```
void CAsyncIo::ProcessRequests(void)
{
    // lock to get the item and increment the outstanding count
    CAsyncRequest * preq = NULL;
    for(;;)
```

```
    {
        {
            CAutoLock lock(&m_csLists);
            preq = GetWorkItem(); // 取出一个数据读取的请求
            if(preq == NULL)
            {
                // done
                return;
            }
            // one more item not on the done or work list
            m_cItemsOut++;
            // release critsec
        }
        preq->Complete();         // 执行实际的数据读取以完成一个数据请求
        // regain critsec to replace on done list
        {
            CAutoLock l(&m_csLists);
    // 将完成的请求放到m_listDone列表中
        // PutDoneItem内部会将m_evDone事件置为有效状态
            PutDoneItem(preq);
            if(--m_cItemsOut == 0)
            {
                if(m_bWaiting)
                    m_evAllDone.Set();
            }
        }
    }
}
```

下面，再来看同步模式下的数据读取函数 CAsyncIo::SyncRead 的实现（注意，并没有生成一个请求放到 m_listWork 列表中，而是“阻塞地”等待当前请求执行完成后再返回）：

```
HRESULT CAsyncIo::SyncRead(
                LONGLONG llPos,
                LONG lLength,
                BYTE * pBuffer)
{
    if(IsAligned(llPos) &&
        IsAligned(lLength) &&
        IsAligned((LONG) pBuffer))
    {
        LONG cbUnused;
        return SyncReadAligned(llPos, lLength, pBuffer, &cbUnused, NULL);
    }
    // not aligned with requirements - use buffered file handle.
    //!!! might want to fix this to buffer the data ourselves?
    CAsyncRequest request;
    HRESULT hr = request.Request(this,
                            m_pStream,
```

```
                                llPos,
                                lLength,
                                FALSE,
                                pBuffer,
                                NULL,
                                0);
        if(FAILED(hr))
        {
            return hr;
        }
        // 等待当前请求执行完成后返回
        return request.Complete();
    }
```

CAsyncOutputPin 类实现了一个输出 Pin，它从 CBasePin 类派生，另外还实现了 IAsyncReader 接口（这是对拉模式的 Source Filter 的基本要求）。IAsyncReader 的所有接口方法如表 12.1 所示。

表 12.1　IAsyncReader 的接口方法及描述

接口方法	描述
BeginFlush	放弃所有正在进行的数据读取
EndFlush	与 BeginFlush 配对，标示 Flush 过程结束
Length	得到数据总长度和当前可以读取的长度
RequestAllocator	要求一个输出 Pin 上的 Sample 管理器
Request	发出一个数据请求（用于异步模式）
SyncReadAligned	同步读取数据（边界对齐）
SyncRead	同步读取数据
WaitForNext	等待一个请求的完成（用于异步模式）

可以发现，CAsyncOutputPin 类上实现的 IAsyncReader 的各个接口方法，都“委托”给了 CAsyncIo 类对象的同名成员函数。

CAsyncReader 类实现了一个 Filter，它从 CBaseFilter 派生，实现了仅含有一个输出 Pin（CAsyncOutputPin 类的实例）的 Source Filter 框架。另外，CAsyncReader 类的构造函数中，一个参数是 CAsyncStream 类型的指针，它是 Filter 获取数据的“源”。

接下去，再来介绍一下 async 项目的实现。这个项目定义了两个新的类：CMemStream 和 CasyncFilter，如图 12.3 所示。

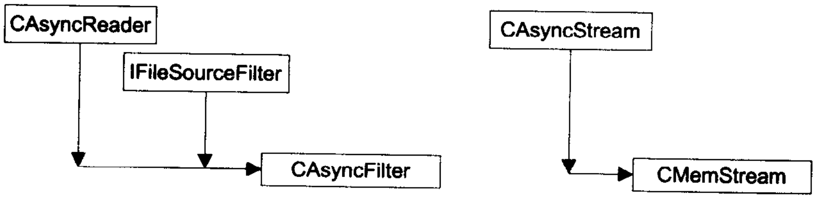

图 12.3　async 项目的类继承关系

CMemStream 类主要实现了 CAsyncStream 类的各个成员函数（因为 CAsyncStream 是

一个纯虚类），它指定了数据源来自一块内存区域。

CAsyncFilter 类定义了 Filter 框架：CLSID 为 CLSID_AsyncSample，注册名为 Sample File Source (Async.)，Merit 值为 MERIT_UNLIKELY 等。在 Filter 上还实现了 IFileSourceFilter 接口，用于源文件的加载（这使得它更像标准的 File Source (Async.) 了）。下面，来看 IFileSourceFilter::Load 函数的实现（它将指定源文件的内容全部读到内存中，然后使用这块数据内存地址来初始化 CMemStream 类的一个对象实例 m_Stream）：

```
STDMETHODIMP Load(LPCOLESTR lpwszFileName, const AM_MEDIA_TYPE *pmt)
{
    CheckPointer(lpwszFileName, E_POINTER);
    // lstrlenW is one of the few Unicode functions that works on win95
    int cch = lstrlenW(lpwszFileName) + 1;
#ifndef UNICODE
    TCHAR *lpszFileName=0;
    lpszFileName = new char[cch * 2];
    if (!lpszFileName) {
          return E_OUTOFMEMORY;
    }
    WideCharToMultiByte(GetACP(), 0, lpwszFileName, -1,
                        lpszFileName, cch, NULL, NULL);
#else
    TCHAR lpszFileName[MAX_PATH]={0};
    lstrcpy(lpszFileName, lpwszFileName);
#endif
    CAutoLock lck(&m_csFilter);
    /*  Check the file type */
    CMediaType cmt;
    if (NULL == pmt) {
        // 设置默认的输出 Pin 上连接使用的媒体类型
        cmt.SetType(&MEDIATYPE_Stream);
        cmt.SetSubtype(&MEDIASUBTYPE_NULL);
    } else {
        cmt = *pmt;
    }
    // 将指定的源文件内容读入内存中，内存指针保存到 m_pbData 中
    if (!ReadTheFile(lpszFileName)) {
#ifndef UNICODE
        delete [] lpszFileName;
#endif
        return E_FAIL;
    }
    // 初始化 CMemStream 类的实例，指定了数据源内存地址，以及有效数据长度
    m_Stream.Init(m_pbData, m_llSize);
    m_pFileName = new WCHAR[cch];
    // 保存源文件名
    if (m_pFileName!=NULL)
         CopyMemory(m_pFileName, lpwszFileName, cch*sizeof(WCHAR));
    // this is not a simple assignment... pointers and format
```

```
    // block (if any) are intelligently copied
    m_mt = cmt;
    /*  Work out file type */
    cmt.bTemporalCompression = TRUE;
    cmt.lSampleSize = 1;
    return S_OK;
}
```

再来看 IFileSourceFilter::GetCurFile 函数的实现（外部调用该函数取得当前的源文件名，以及输出 Pin 上的媒体类型）：

```
STDMETHODIMP GetCurFile(LPOLESTR * ppszFileName, AM_MEDIA_TYPE *pmt)
{
    CheckPointer(ppszFileName, E_POINTER);
    *ppszFileName = NULL;
    if (m_pFileName!=NULL) {
        DWORD n = sizeof(WCHAR)*(1+lstrlenW(m_pFileName));
        *ppszFileName = (LPOLESTR) CoTaskMemAlloc( n );
        if (*ppszFileName!=NULL) {
              CopyMemory(*ppszFileName, m_pFileName, n);
        }
    }
    if (pmt!=NULL) {
        CopyMediaType(pmt, &m_mt);
    }
    return NOERROR;
}
```

提示　为了深入理解拉模式 Source Filter 的实现，请读者同时参考 SDK 基类源代码中的 CPullPin 类的实现（pullpin.cpp）。

12.2　推模式例子

这一节，将介绍一个推模式的 Source Filter 的实现，它位于 SDK 安装目录 Samples\C++\DirectShow\Filters\Ball 下。

12.2.1　实现的功能

Filter 的注册名：“Bouncing Ball”。

Filter 类型：Source Filter（进含一个输出 Pin）。

CLSID：CLSID_BouncingBall。

Merit 值：MERIT_DO_NOT_USE。

功能描述：模拟 Live Source 不断产生视频帧，演示在封闭围墙内碰撞的弹球的运动轨迹。输出视频格式包括 RGB32、RGB24、RGB565、RGB555、RGB8 等，并且弹球的颜色根据输出 Pin 上连接使用的媒体类型的不同而不同。

使用 Bouncing Ball Filter 构建 Filter Graph 及其输出结果如图 12.4 和图 12.5 所示。

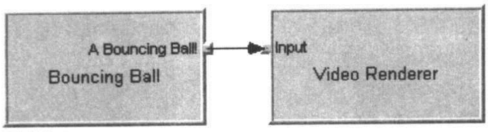

图 12.4 使用 Bouncing Ball Filter 构建的 Filter Graph

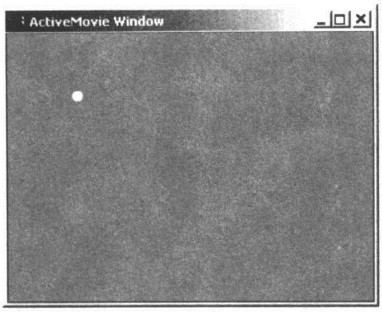

图 12.5 Bouncing Ball Filter 的输出结果

12.2.2 实现要点

Filter 的类名为 CBouncingBall，Pin 的类名为 CBallStream。它们的继承关系如图 12.6 所示。

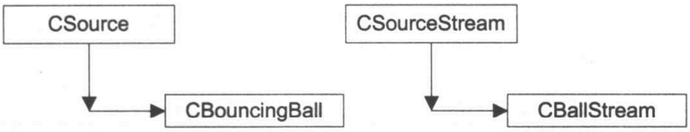

图 12.6 Bouncing Ball Filter 的类继承关系

可以知道，从 CSource 派生的 Filter，它的输出 Pin 会使用一个数据线程，将 Sample 不断地“推”出去。具体的过程可以参见 CSourceStream::DoBufferProcessingLoop 的实现：

```
HRESULT CSourceStream::DoBufferProcessingLoop(void)
{
    Command com;
    OnThreadStartPlay(); // 开始数据传送前调用，子类可以重新实现以进行一些初始化
    do {
   while (!CheckRequest(&com)) {
       IMediaSample *pSample;
           // 输出 Pin 上得到一个空的 Sample
       HRESULT hr = GetDeliveryBuffer(&pSample,NULL,NULL,0);
       if (FAILED(hr)) {
               Sleep(1);
          continue;    // go round again. Perhaps the error will go away
                     // or the allocator is decommited & we will be asked
                     // to exit soon.
       }
       // 子类必须实现的一个函数，填写 Sample 的实际内容
```

```
        hr = FillBuffer(pSample);
        if (hr == S_OK) {
           hr = Deliver(pSample);   // 往下发送 Sample
                 pSample->Release();
                 // downstream filter returns S_FALSE if it wants us to
                 // stop or an error if it's reporting an error.
                 if(hr != S_OK)
                 {
                    DbgLog((LOG_TRACE, 2, TEXT("Deliver() returned %08x;
                          stopping"), hr));
                    return S_OK;
                 }
        } else if (hr == S_FALSE) {
                 // derived class wants us to stop pushing data
           pSample->Release();
           DeliverEndOfStream();
           return S_OK;
        } else {
                 // derived class encountered an error
                 pSample->Release();
          DbgLog((LOG_ERROR, 1, TEXT("Error %08lX from FillBuffer!!!"),
                hr));
                 DeliverEndOfStream();
                 m_pFilter->NotifyEvent(EC_ERRORABORT, hr, 0);
                 return hr;
        }
             // all paths release the sample
    }
        // For all commands sent to us there must be a Reply call!
    if (com == CMD_RUN || com == CMD_PAUSE) {
        Reply(NOERROR);
    } else if (com != CMD_STOP) {
        Reply((DWORD) E_UNEXPECTED);
        DbgLog((LOG_ERROR, 1, TEXT("Unexpected command!!!")));
    }
    } while (com != CMD_STOP);

    return S_FALSE;
}
```

CBallStream::FillBuffer 的实现如下：

```
HRESULT CBallStream::FillBuffer(IMediaSample *pms)
{
    CheckPointer(pms,E_POINTER);
    ASSERT(m_Ball);
    BYTE *pData;
    long lDataLen;
    pms->GetPointer(&pData);
    lDataLen = pms->GetSize();
    ZeroMemory(pData, lDataLen);  // 清屏：将图像帧填为黑色
```

```
    {
        CAutoLock cAutoLockShared(&m_cSharedState);
        // If we haven't just cleared the buffer delete the old
        // ball and move the ball on
        m_Ball->MoveBall(m_rtSampleTime - (LONG) m_iRepeatTime);
        // 在图像帧上绘出弹球当前所在的位置
        m_Ball->PlotBall(pData, m_BallPixel, m_iPixelSize);
        // The current time is the sample's start
        CRefTime rtStart = m_rtSampleTime;
        // Increment to find the finish time
        m_rtSampleTime += (LONG)m_iRepeatTime;
        // 为 Sample 打上时间戳
        pms->SetTime((REFERENCE_TIME *) &rtStart,(REFERENCE_TIME *)
                    &m_rtSampleTime);
    }
    pms->SetSyncPoint(TRUE);  // 因为图像帧是非压缩的，任何帧都可以是同步点
    return NOERROR;
}
```

可以看到，m_Ball 是一个 CBall 类（参见 ball.cpp 的实现）的实例，主要完成弹球的位置计算以及绘制。当要在一个新位置上绘制弹球时，一般首先调用 CBall::MoveBall 进行新位置的计算，然后调用 CBall::PlotBall 在计算得到的新位置上绘制一个弹球。（绘制的过程，实际上也就是在图像帧内将特定位置的像素值换成代表弹球的像素值。）

理解 Bouncing Ball 这个例子，还需注意以下几点：

- 如何创建输出 Pin。因为 Filter–Pin 采用 CSource–CSourceStream 类结构，所以实际输出 Pin 的创建一般放在 Filter 的构造函数中，如下：

```
CBouncingBall::CBouncingBall(LPUNKNOWN lpunk, HRESULT *phr) :
    CSource(NAME("Bouncing ball"), lpunk, CLSID_BouncingBall)
{
    ASSERT(phr);
    CAutoLock cAutoLock(&m_cStateLock);
    // m_paStreams 是一个指针数组，因为只有一个输出 Pin，所以以下分配一个单元
    m_paStreams = (CSourceStream **) new CBallStream*[1];
    if(m_paStreams == NULL)
    {
        if(phr)
            *phr = E_OUTOFMEMORY;
        return;
    }
    //新生成一个 CBallStream 的实例
    m_paStreams[0] = new CBallStream(phr, this, L"A Bouncing Ball!");
    if(m_paStreams[0] == NULL)
    {
        if(phr)
            *phr = E_OUTOFMEMORY;
        return;
    }
}
```

提示　在 Filter 退出时，Pin 对象的释放工作已经由父类完成。在 CBouncingBall 中可以不考虑这个问题。

- 如何在输出 Pin 上提供首选的媒体类型列表，包括 RGB32、RGB24、RGB565、RGB555、RGB8 等。参见 CBallStream::GetMediaType 的实现。
- 如何在输出 Pin 连接时进行媒体类型检查，参见 CBallStream::CheckMediaType。
- 当输出 Pin 连接完成，需要协商 Pin 上以后传送数据使用的 Sample 的一些特性，比如 Sample 使用的内存有多大。按照惯例，传送非压缩视频图像的 Sample 大小就取为一帧图像的大小。参见 CBallStream::DecideBufferSize 的实现。
- 如何实现根据输出 Pin 上的媒体类型变换弹球的颜色。可以看到，CBallStream 类有一个成员变量 m_BallPixel，记录的就是当前使用的弹球颜色。这个变量的取值在 CBallStream::SetMediaType 函数中根据实际连接使用的媒体类型而定。CBall::PlotBall 函数的第 2 个参数就是指弹球的颜色值，第 3 个参数为每个像素的字节数。
- 如何支持 RGB8 格式。大家知道，RGB8 是调色板格式，图像帧中每个像素的颜色用一个字节表示，只是这个字节不是颜色值，而是调色板中的索引值。因为 10 是系统调色板中的第一个非保留索引，可以将这个位置的颜色值指定为需要的颜色，然后在绘制弹球时将弹球的像素值都替换为 10。注意，在 CBallStream::GetMediaType 中提供 RGB8 的媒体类型时，需要同时制定调色板颜色值：

```
pvi->bmiHeader.biClrUsed = iPALETTE_COLORS;
for(int i = 0; i < iPALETTE_COLORS; i++)
{
    pvi->TrueColorInfo.bmiColors[i].rgbRed = m_Palette[i].peRed;
    pvi->TrueColorInfo.bmiColors[i].rgbBlue = m_Palette[i].peBlue;
    pvi->TrueColorInfo.bmiColors[i].rgbGreen = m_Palette[i].peGreen;
    pvi->TrueColorInfo.bmiColors[i].rgbReserved = 0;
}
```

- 如何给 Sample 打时间戳。时间戳一般应该从 0 开始打起，因此可以看到 CBallStream::OnThreadCreate 中将 m_rtSampleTime 初始化为 0。CBallStream 类还定义了一个变量 m_iRepeatTime，表示单帧图像显示的时间。在整个 Filter 的运行过程中，时间戳应该是连续的，因此 m_rtSampleTime 会在 CBallStream::FillBuffer 中累计发出去 Sample 的结束时间点，以便作为下一个 Sample 的开始时间。
- 如何响应质量控制事件。需要在输出 Pin 上实现 Notify 函数：

```
STDMETHODIMP CBallStream::Notify(IBaseFilter * pSender, Quality q)
{
    // 调整 Sample 帧的单帧显示时间，也就是调整 Source Filter 发送 Sample 的频率
    if(q.Proportion<=0)
    {
        m_iRepeatTime = 1000;    // We don't go slower than 1 per second
    }
```

```
    else
    {
        m_iRepeatTime = m_iRepeatTime*1000 / q.Proportion;
        if(m_iRepeatTime>1000)
        {
            m_iRepeatTime = 1000; // We don't go slower than 1 per second
        }
        else if(m_iRepeatTime<10)
        {
            m_iRepeatTime = 10;      // We don't go faster than 100/sec
        }
    }
    // 如果到达 Video Renderer 的 Sample 晚了 q.Late,
    // 就在下一个发出去的 Sample 的时间戳上加上这个偏移量
    if(q.Late > 0)
        m_rtSampleTime += q.Late;
    return NOERROR;
}
```

第 13 章　Transform Filter 例子

13.1　Transform 例子

这一节，将介绍一个普通的 Transform Filter 的实现，它位于 SDK 安装目录 Samples\C++\DirectShow\Filters\Contrast 下。

13.1.1　实现的功能

Filter 的注册名：“Video Contrast”。

Filter 类型：Transform Filter（一个输入 Pin、一个输出 Pin）。

CLSID：CLSID_Contrast。

Merit 值：MERIT_DO_NOT_USE。

功能描述：调整视频图像的对比度。仅支持 RGB8 格式的输入和输出。

使用 Video Contrast 构建 Filter Graph 如图 13.1 所示。

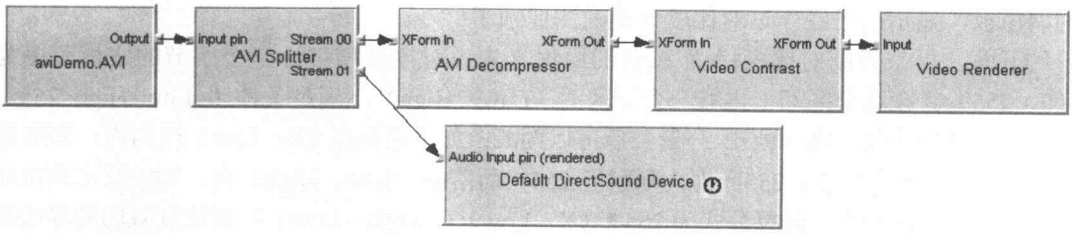

图 13.1　使用 Video Contrast 构建的 Filter Graph

上述 Filter Graph 在运行过程中，可以通过 Video Contrast Filter 的属性页，动态地修改输入图像的对比度，如图 13.2 所示。

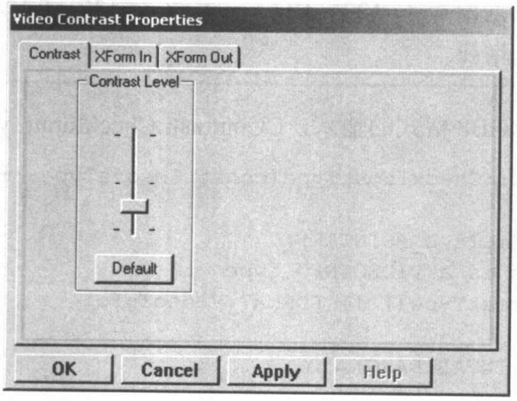

图 13.2　Video Contrast Filter 的属性页

13.1.2 实现要点

Filter 的类名为 CContrast（没有实现自定义的 Pin 类），属性页的类名为 Ccontrast-Properties。它们的继承关系如图 13.3 所示。

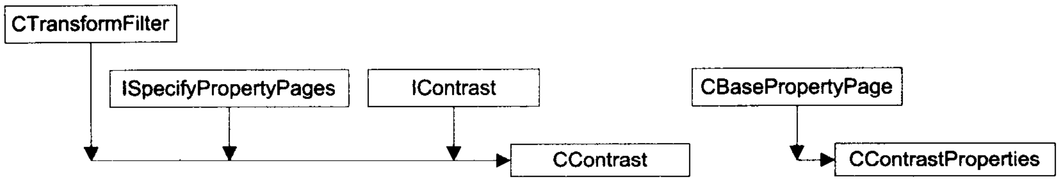

图 13.3 CContrast 和 CContrastProperties 的类继承关系

这个 Filter 仅接受 RGB8 格式的输入媒体类型。另外，Filter 在整个数据处理过程中并没有改变输入 Sample 的实际内容。那么这个 Filter 是如何实现输入图像的对比度调整的呢？

可以知道，RGB8 格式使用调色板来描述图像帧（调色板实际上是存储一系列 RGB 颜色值的数组），一个字节表示一个像素，但这个字节表示的是当前像素使用的颜色值在调色板中对应的索引值。既然没有在 Filter 中改变图像帧的内容，那么可以通过改变调色板中的颜色值，变相地影响最终“解释”出来的图像帧。实际上，当将这个改变了的调色板（携带在媒体类型中），跟着 Sample 一起发送下去后，Video Renderer 就会根据这个新的调色板，显示出已经作过对比度调整的图像帧了。

下面，再来介绍对比度增强/减弱的原理。先说对比度增强。一个字节的取值范围为（0，255）。在这个区间上再取一个子区间（Low，High），满足条件 0<Low<High<255。于是，得到一个像素值（RGB 分量）后可以判断：如果它落在（0，Low）区间内，则取值为 0；如果落在（High，255）内，则取值 255；而落在（Low，High）内，则按照比例拉伸到（0，255）区间上，计算公式为 255 * （X – Low） / （High – Low）。对比度减弱的算法也很简单，它让目标像素值在一个均值（对比度绝对值）的基础上上下浮动；当对比度减弱程度最大时，图像帧完全变成灰色。如果用 L 表示对比度绝对值，那么对比度减弱后的像素值计算公式为 L + X * (255 – 2 * L) / 255。

提示 Filter 对比度取值范围为（-127，127），其中（-127，0）表示对比度减弱，（0，127）表示对比度增强。

既然 Filter 仅接受 RGB8 格式的输入，CContrast::CheckInputType 的实现如下：

```
HRESULT CContrast::CheckInputType(const CMediaType *mtIn)
{
    CheckPointer(mtIn,E_POINTER);
    // Check this is a VIDEOINFO type
    if(*mtIn->FormatType() != FORMAT_VideoInfo)
    {
        return E_INVALIDARG;
    }
    // 进一步检查这种媒体类型是否支持对比度调整（这个 Filter 仅接受 RGB8）
    if(CanChangeContrastLevel(mtIn))
```

```
    {
        return NOERROR;
    }
    return E_FAIL;
}
 BOOL CContrast::CanChangeContrastLevel(const CMediaType *pMediaType)
const
{
    CheckPointer(pMediaType,FALSE);
    if((IsEqualGUID(*pMediaType->Type(), MEDIATYPE_Video)) &&
       (IsEqualGUID(*pMediaType->Subtype(), MEDIASUBTYPE_RGB8)))
    {
        // I think I can process this format (8 bit palettised)
        // So do a quick sanity check on the palette information
        VIDEOINFO *pvi = (VIDEOINFO *) pMediaType->Format();
        return (pvi->bmiHeader.biBitCount == 8);//最后确认每个像素使用的 Bit 数
    }
    else
    {
        return FALSE;
    }
}
```

另外，输出 Pin 上的媒体类型应该保持与输入 Pin 上的媒体类型完全一致：

```
HRESULT CContrast::CheckTransform(const CMediaType *mtIn,const
                                  CMediaType *mtOut)
{
    CheckPointer(mtIn,E_POINTER);
    CheckPointer(mtOut,E_POINTER);
    HRESULT hr;
    if(FAILED(hr = CheckInputType(mtIn)))
    {
        return hr;
    }
    // 格式细节必须使用数据结构 VIDEOINFOHEADER 来描述
    if(*mtOut->FormatType() != FORMAT_VideoInfo)
    {
        return E_INVALIDARG;
    }
    // formats must be big enough
    if(mtIn->FormatLength() < sizeof(VIDEOINFOHEADER) ||
          mtOut->FormatLength() < sizeof(VIDEOINFOHEADER))
        return E_INVALIDARG;
    VIDEOINFO *pInput  = (VIDEOINFO *) mtIn->Format();
    VIDEOINFO *pOutput = (VIDEOINFO *) mtOut->Format();
    // 比较两个媒体类型的 BITMAPINFOHEADER 数据结构描述，必须完全一致！
          if(memcmp(&pInput->bmiHeader,&pOutput-
    >bmiHeader,sizeof(BITMAPIN-
      FOHEADER)) == 0)
    {
```

```
        return NOERROR;
    }
    return E_INVALIDARG;
}
```

下面再来看对比度调整的实现：

```
HRESULT CContrast::Transform(IMediaSample *pIn, IMediaSample *pOut)
{
    HRESULT hr = Copy(pIn, pOut);
    if (FAILED(hr)) {
        return hr;
    }
    return Transform(pOut);
}
```

其中，CContrast::Copy(IMediaSample *pSource, IMediaSample *pDest)函数实现了输入 Sample 到输出 Sample 的完整拷贝，CContrast::Transform(IMediaSample *pMediaSample)实现了输出 Sample 的对比度调整：

```
HRESULT CContrast::Transform(IMediaSample *pMediaSample)
{
    CheckPointer(pMediaSample,E_POINTER);
    signed char ContrastLevel;
    // Take a copy, so we dont hold the lock for the whole transform.
    // Also ensures that we use the same level throughout this!
    {
        CAutoLock cAutoLock(&m_ContrastLock);
        ContrastLevel = m_ContrastLevel;
    }
    AM_MEDIA_TYPE *pAdjustedType = NULL;
    pMediaSample->GetMediaType(&pAdjustedType);
    if(pAdjustedType != NULL)
{
    // 安全起见，再次检查媒体类型
        if(CheckInputType(&CMediaType(*pAdjustedType)) == S_OK)
        {
            m_pInput->CurrentMediaType() = *pAdjustedType;
            CoTaskMemFree(pAdjustedType);
        }
        else
        {
            CoTaskMemFree(pAdjustedType);
            return E_FAIL;
        }
    }
    // Pass on format changes to downstream filters
    if((pAdjustedType != NULL) || (m_PrevLevel != ContrastLevel))
    {
        CMediaType AdjustedType((AM_MEDIA_TYPE) m_pInput->Current-
                                MediaType());
```

```
        // 改变媒体类型的调色板部分
        HRESULT hr = Transform(&AdjustedType, ContrastLevel);
        if(hr == S_OK)
        {
            // 将改变后的媒体类型设置在输出的 Sample 上，以后随 Sample 一起发送出去
            pMediaSample->SetMediaType(&AdjustedType);
            m_PrevLevel = ContrastLevel;
        }
        else
        {
            return hr;
        }
    }
    return NOERROR;
}
// 实现了媒体类型中调色板颜色的变换
HRESULT CContrast::Transform(AM_MEDIA_TYPE *pType, const signed char
                             ContrastLevel) const
{
    CheckPointer(pType,E_POINTER);
    VIDEOINFO *pvi = (VIDEOINFO *) pType->pbFormat;
    if(ContrastLevel >= 0)
    {
        int Low    = 0 + m_ContrastLevel;     // 计算一个子区间
        int High   = 255 - m_ContrastLevel;
        float Grad = ((float)(High - Low)) / 255;
        for(UINT i = 0; i < pvi->bmiHeader.biClrUsed; i++)
        {
            // 增加对比度
            IncreaseContrast(&pvi->bmiColors[i], Low, High, Grad);
        }
    }
    else
    {
        float Grad = 255 / (255 + (float) ContrastLevel + (float)
                        ContrastLevel);
        for(UINT i = 0; i < pvi->bmiHeader.biClrUsed; i++)
        {
            // 减弱对比度
            DecreaseContrast(&pvi->bmiColors[i], ContrastLevel, Grad);
        }
    }
    return NOERROR;
}
```

对比度增强/减弱的具体算法实现如下：

```
// 增强对比度
inline
void CContrast::IncreaseContrast(RGBQUAD *pElem, const int Low, const
                                 int High,const float Grad) const
```

```
{
    ASSERT(pElem);
    // 分别取 RGB 各分量做对比度增强处理
    IncreaseContrast(&pElem->rgbRed, Low, High, Grad);
    IncreaseContrast(&pElem->rgbGreen, Low, High, Grad);
    IncreaseContrast(&pElem->rgbBlue, Low, High, Grad);
}
// 真正的对比度增强算法实现
inline
void CContrast::IncreaseContrast(BYTE *pByte, const int Low, const int
                                 High,const float Grad) const
{
    ASSERT(pByte);
    if(*pByte <= Low)
    {
        *pByte = 0;
    }
    else if((Low < *pByte) && (*pByte <  High))
    {
        *pByte = (BYTE)( (*pByte - Low) / Grad);
    }
    else
    {  // pElem->rgbGreen >= High
        *pByte = 255;
    }
}
// 减弱对比度
inline
void CContrast::DecreaseContrast(RGBQUAD *pElem, const int Level, const
                                 float Grad) const
{
    ASSERT(pElem);
    // 分别取 RGB 各分量做对比度减弱处理
    DecreaseContrast(&pElem->rgbRed, Level, Grad);
    DecreaseContrast(&pElem->rgbGreen, Level, Grad);
    DecreaseContrast(&pElem->rgbBlue, Level, Grad);
}
// 真正减弱对比度的算法实现
inline
void CContrast::DecreaseContrast(BYTE *pByte, const int Level, const
                                 float Grad) const
{
    ASSERT(pByte);
    ASSERT(Grad != 0.0);
    *pByte = (BYTE) ( ((int) (*pByte / Grad)) - Level);
}
```

提示 上面几个函数都使用了内联（inline）关键字，以提高程序执行的速度。

为了让 Filter 外部可以调整对比度，这个 Filter 还实现了一个自定义接口 IContrast，它的定义如下：

```
#ifdef __cplusplus
extern "C" {
#endif
    // {fd5010a2-8ebe-11ce-8183-00aa00577da1}
    DEFINE_GUID(IID_IContrast,0xfd5010a2, 0x8ebe, 0x11ce, 0x81, 0x83,
    0x00, 0xaa, 0x00, 0x57, 0x7d, 0xa1);
    DECLARE_INTERFACE_(IContrast, IUnknown)
    {
        STDMETHOD(get_ContrastLevel) (THIS_
            signed char *ContrastLevel       // 得到当前的对比度值
        ) PURE;
        STDMETHOD(put_ContrastLevel) (THIS_  // 设置新的对比度值
            signed char ContrastLevel,
            unsigned long ChangeTime
        ) PURE;
        STDMETHOD(put_DefaultContrastLevel) (THIS) PURE;
                                        // 设置使用默认的对比度值
                                        // 默认值为 0，表示对比度不做任何调整
    };
#ifdef __cplusplus
}
#endif
```

为了方便自定义接口的调试，这个 Filter 还实现了一个属性页（参见 ContProp.cpp 文件）。Filter 属性页的实现，最主要的还是要熟悉 Windows API 编程。这里不再展开。

13.2　Trans-In-Place 例子

这一节，将介绍一个“就地”处理的 Transform Filter 的实现，它位于 SDK 安装目录 Samples\C++\DirectShow\Filters\NullInPlace 下。

13.2.1　实现的功能

Filter 的注册名：“Null In Place”；

Filter 类型：Trans-In-Place Filter（一个输入 Pin、一个输出 Pin）；

CLSID：CLSID_NullInPlace；

Merit 值：MERIT_DO_NOT_USE；

功能描述：枚举连接着的上一级 Filter 的输出 Pin 上支持的所有媒体类型，可以指定其中一种，用于与下一级 Filter 的连接。

使用 Null In Place 构建 Filter Graph 如图 13.4 所示。

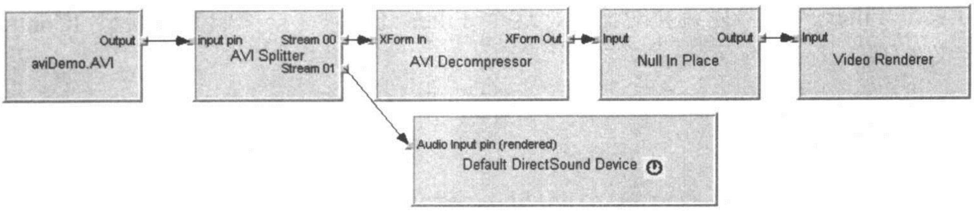

图 13.4 使用 Null In Place 构建的 Filter Graph

如图 13.5，Null In Place Filter 的属性页中列出了 AVI Decompressor 的输出 Pin 能够输出的所有媒体类型。如果不指定任何一种类型，那么 AVI Decompressor 与 Video Renderer 的连接将（随机）选择其中一种双方都能接受的媒体类型。如果指定了一种媒体类型（当然也要 AVI Decompressor 与 Video Renderer 都能接受），这两个 Filter 之间的连接将只使用这种媒体类型。

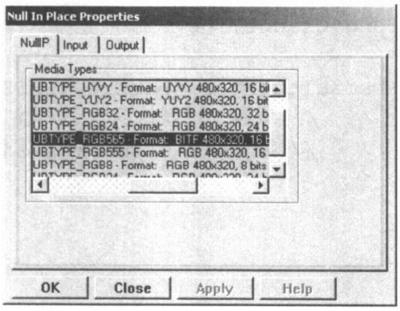

图 13.5 Null In Place 的属性页

13.2.2 实现要点

NullInPlace 项目实现了很多新的类，包括 CNullInPlace（Filter 类）、CNullInPlace-InputPin（输入 Pin 类）、CNullInPlaceOutputPin（输出 Pin 类）、NullIPProperties（Filter 的属性页类）、CTextMediaType（用于将媒体类型字符化的工具类）等。这些类的继承结构如图 13.6 所示。

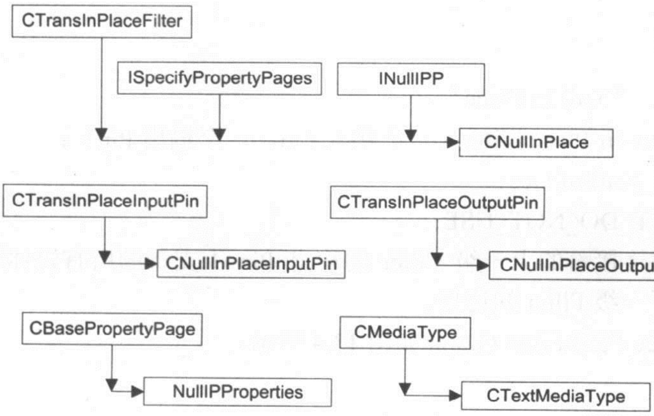

图 13.6 NullInPlace 项目中定义的类结构

下面，再来看一下自定义接口 INullIPP 的定义：

```
DECLARE_INTERFACE_(INullIPP, IUnknown)
{
    // 指定一个连接用的媒体类型
    STDMETHOD(put_MediaType) (THIS_
              CMediaType *pmt /* [in] */ // the media type selected
              ) PURE;
    // 得到指定的媒体类型
    STDMETHOD(get_MediaType) (THIS_
              CMediaType **pmt    /* [out] */ // the media type selected
              ) PURE;
    // 得到连接着的上一级 Filter 的输出 Pin
    STDMETHOD(get_IPin) (THIS_
              IPin **pPin         /* [out] */ // the source pin
              ) PURE;
    // 得到当前的 Filter 状态
    STDMETHOD(get_State) (THIS_
              FILTER_STATE *state  /* [out] */ // the filter state
              ) PURE;
};
```

可以看到，Null In Place 是一个标准的 Trans-In-Place Filter。外部可以通过自定义接口 INullIPP 的接口方法，指定一种 Filter 的输入和输出 Pin 连接使用的媒体类型。于是，Filter 的输入和输出 Pin 均重新实现了 CheckMediaType 函数，以在连接时对连接用的媒体类型与指定的媒体类型进行对比：

```
// 输入 Pin 的 CheckMediaType 函数实现
HRESULT CNullInPlaceInputPin::CheckMediaType( const CMediaType *pmt )
{
    CNullInPlace *pNull = (CNullInPlace *) m_pTIPFilter;
    CheckPointer(pNull,E_UNEXPECTED);
#ifdef DEBUG
    DisplayType(TEXT("Input type proposed"),pmt);
#endif
// 如果外部没有指定媒体类型，则遵循父类的实现
    if (pNull->m_mtPreferred.IsValid() == FALSE)
    {
        if( pNull->m_pOutput->IsConnected() ) {
            return pNull->m_pOutput->GetConnected()->QueryAccept( pmt );
        }
        return S_OK;
    }
    else  // 如果外部指定了媒体类型，则将当前试连接用的媒体类型跟它作比较
        if (*pmt == pNull->m_mtPreferred)
            return S_OK  ;
        else
            return VFW_E_TYPE_NOT_ACCEPTED;
}
// 输出 Pin 的 CheckMediaType 函数实现
```

```
HRESULT CNullInPlaceOutputPin::CheckMediaType( const CMediaType *pmt )
{
    CNullInPlace *pNull = (CNullInPlace *) m_pTIPFilter;
    CheckPointer(pNull,E_UNEXPECTED);
    // 如果外部没有指定媒体类型，则遵循父类的实现
    if (pNull->m_mtPreferred.IsValid() == FALSE)
    {
        return CTransInPlaceOutputPin::CheckMediaType (pmt) ;
    }
    else
{
    // 如果外部指定了媒体类型，则将当前试连接用的媒体类型跟它作比较
        if (*pmt == pNull->m_mtPreferred)
            return S_OK  ;
        else
            return VFW_E_TYPE_NOT_ACCEPTED;
    }
}
```

可以看到，m_mtPreferred 是 Filter 类上的一个 CMediaType 类型的成员对象，用以保存外部指定的媒体类型。m_mtPreferred 在 Filter 的构造函数中调用了 m_mtPreferred.InitMediaType()，表示默认情况下，Filter 没有被指定任何有效的媒体类型。

下面再来看 INullIPP 的接口方法 get_IPin 的实现（要先取得连接着的上一级 Filter 的输出 Pin，然后才能枚举它上面支持的所有媒体类型）：

```
STDMETHODIMP CNullInPlace::get_IPin (IPin **ppPin)
{
    CheckPointer(ppPin,E_POINTER);
    CAutoLock l(&m_NullIPLock);
    if (!m_pInput)
    {
        *ppPin = NULL;
        return NOERROR;
    }
    // 判断输入 Pin 是否连接
    if (!m_pInput->IsConnected())
    {
        *ppPin = NULL;
    }
    else
    {
    // 得到与输入 Pin 连接着的上一级 Filter 的输出 Pin
        *ppPin = m_pInput->GetConnected ();
        // 增加一个引用计数，外部使用完这个 Pin 后，应该调用 Release
        (*ppPin)->AddRef();
    }
    return NOERROR ;
}
```

至于 INullIPP 各接口方法的使用方法，可以参见 Null In Place Filter 的属性页的实

现。这个属性页的实现中，最主要的还是 Pin 上媒体类型的枚举（参见 NullIPProperties::FillListBox 函数的实现）。这里不再展开。

最后，介绍一下跟踪“同一个 Filter 的多个实例”的问题。已知 Filter 类中定义了一个静态变量“static m_nInstanceCount;”，用于表示这个 Filter 被创建的实例总数量（它在 Filter 实现文件第一次被转入内存时初始化为 0，如下：int CNullInPlace::m_nInstanceCount = 0;）；还定义了一个成员变量“int m_nThisInstance;”，用于表示当前实例的索引号。下面来看 Filter 类构造函数的实现：

```
CNullInPlace::CNullInPlace(TCHAR *tszName, LPUNKNOWN punk, HRESULT *phr)
    : CTransInPlaceFilter (tszName, punk, CLSID_NullInPlace, phr)
{
    // 实例计数器加 1 后赋值给当前实例的 m_nThisInstance 变量，表示当前实例的索引号
    m_nThisInstance = ++m_nInstanceCount;
    m_mtPreferred.InitMediaType () ;
    // DEBUG 时输出程序执行进程（即执行到哪个函数内部）
    DbgFunc("CNullInPlace");
}
```

其中，DbgFunc 宏的定义如下：

```
#define DbgFunc(a) DbgLog(( LOG_TRACE                        \
                          , 2                                \
                          , TEXT("CNullInPlace(Instance %d)::%s\0") \
                          , m_nThisInstance                  \
                          , TEXT(a)                          \
                         ));
```

第 14 章　Renderer Filter 例子

14.1　Video Renderer 例子

这一节，将介绍一个 Video Renderer Filter 的实现，它位于 SDK 安装目录 Samples\C++\DirectShow\Filters\SampVid 下。

提示　这个 Video Renderer 例子与 DirectShow 标准的 Video Renderer 相比，功能要简单得多，效率也低很多（单纯使用 GDI 函数画图）。这里主要通过这个例子的学习，理解 Video Renderer 的一般工作过程；但并不推荐将这个 Filter 直接用于商业软件中。

14.1.1　实现的功能

Filter 的注册名：“Sample Video Renderer”；

Filter 类型：Renderer Filter（仅有一个输入 Pin）；

CLSID：CLSID_SampleRenderer；

Merit 值：MERIT_DO_NOT_USE；

功能描述：接收 RGB 格式的视频 Sample，并在不规则窗口中显示出来（功能与 DirectShow 标准的 Video Renderer 类似）。

使用 Sample Video Renderer 构建 Filter Graph 及运行时的视频窗口如图 14.1 所示。

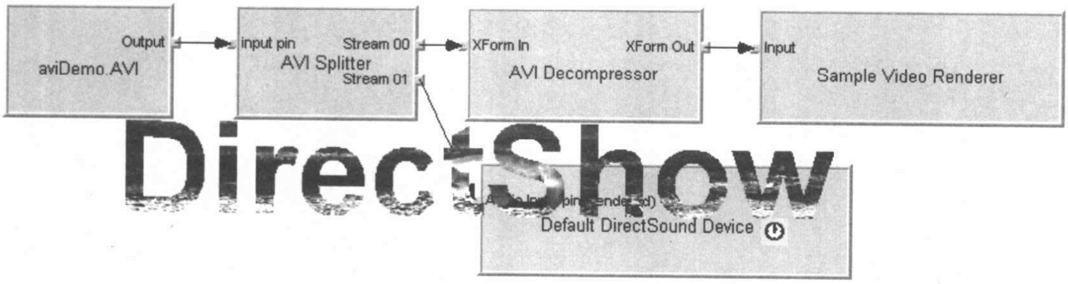

图 14.1　使用 Sample Video Renderer 构建的 Filter Graph 及运行时的视频窗口

如图 14.2，Sample Video Renderer 还实现了一个属性页（与 DirectShow 标准的 Video Renderer 类似），用于标示运行时的各性能参数。

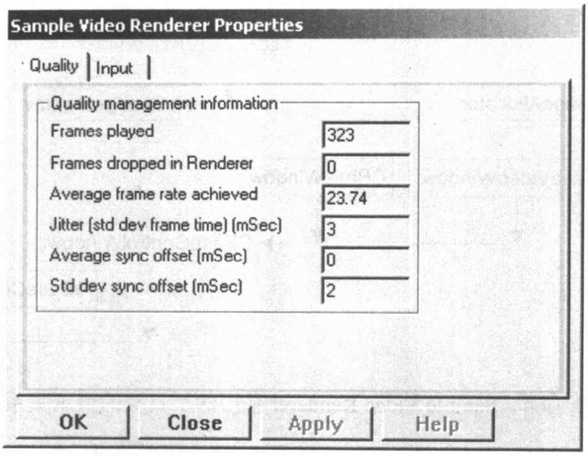

图 14.2 Sample Video Renderer 的属性页

14.1.2 实现要点

Filter 的类名为 CVideoRenderer，Pin 的类名为 CVideoInputPin，属性页类名为 CqualityProperties。这些类的继承关系如图 14.3 所示。

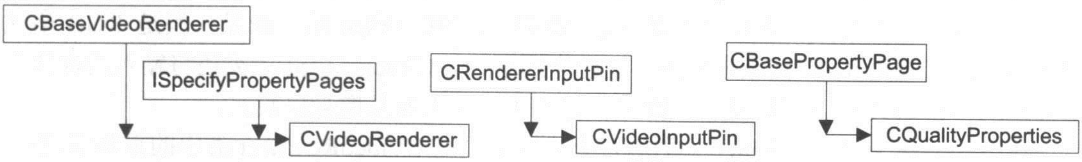

图 14.3 Sample Video Renderer 相关类的继承关系

理解这个 Filter 实现的框架，很大程度上取决于对图 14.3 中几个基类的熟悉程度。这些基类的实现，可以参考 SDK 基类源代码项目中的 renbase.cpp 文件。

下面再来看 CVideoRenderer 类中定义的一些成员变量：

```
CImageAllocator m_ImageAllocator;//Sample 管理器，带有 DIBSECTION 数据结构的定义
CVideoInputPin   m_InputPin;          // 输入 Pin
CImageDisplay    m_Display;           // 主要用于管理图像显示格式
CMediaType       m_mtIn;              // 连接用的媒体类型
CVideoText       m_VideoText;         // （不规则）视频窗口
CImagePalette    m_ImagePalette;      // 主要用于管理调色板
CDrawImage       m_DrawImage;         // 完成实际的画图工作
SIZE             m_VideoSize;         // 当前视频图像大小
```

定义上述成员，使用到了一系列的工具类，它们的继承关系如图 14.4 所示。

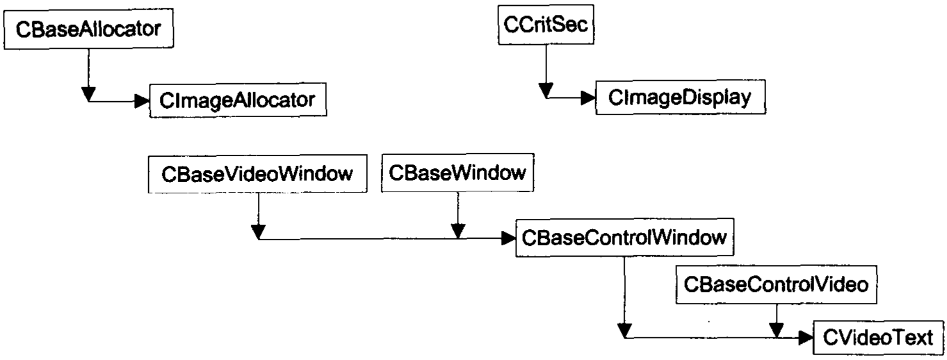

图 14.4 Sample Video Renderer 使用的工具类的继承关系

下面，来分析整个 Filter 框架的实现。先来看媒体类型的检查：

```
HRESULT CVideoRenderer::CheckMediaType(const CMediaType *pmtIn)
{
    return m_Display.CheckMediaType(pmtIn);
}
```

可以看到，媒体类型检查被“委托”到了 CImageDisplay 类的 CheckMediaType 函数；在它的内部实现中，媒体类型要求是 MEDIATYPE_Video 的，而且一般是 RGB 格式的（因为最终是用 GDI 函数来画图）。整个过程中，CImageDisplay 类还将输入的媒体类型描述的图像格式与当前显示器的设置进行了比较（主要是像素位数）。

接下去，再来看一下 Filter 父类的几个成员函数在 CVideoRenderer 类中的重新实现：

```
// 在画图之前，初始化调色板
void CVideoRenderer::PrepareRender()
{
    // Realise the palette on this thread
    m_VideoText.DoRealisePalette();
}
// 将当前的 Sample 显示出来（实际上调用了 CDrawImage::DrawImage 函数）
HRESULT CVideoRenderer::DoRenderSample(IMediaSample *pMediaSample)
{
    return m_DrawImage.DrawImage(pMediaSample);
}
// 当 Filter 接收到第一个 Sample 后调用该函数
// CVideoRenderer 将这个 Sample 显示出来
void CVideoRenderer::OnReceiveFirstSample(IMediaSample *pMediaSample)
{
    ASSERT(pMediaSample);
    DoRenderSample(pMediaSample);
}
// 输入 Pin 试连接用的媒体类型检查通过后，设置给 Filter 保存
HRESULT CVideoRenderer::SetMediaType(const CMediaType *pmt)
{
    CheckPointer(pmt,E_POINTER);
    HRESULT hr = NOERROR;
```

```
    CAutoLock cInterfaceLock(&m_InterfaceLock);
    CMediaType StoreFormat(m_mtIn);
    m_mtIn = *pmt;
    VIDEOINFO *pVideoInfo = (VIDEOINFO *) m_mtIn.Format();
    // 更新图像格式描述
    m_Display.UpdateFormat(pVideoInfo);
    // 新旧调色板的比较，如果两个调色板颜色值一致则不用重新设置
    hr = m_ImagePalette.PreparePalette(&m_mtIn,&StoreFormat,NULL);
    if (FAILED(hr)) {
        return hr;
    }
    // 对其他成员对象进行初始化
    m_DrawImage.NotifyMediaType(&m_mtIn);
    m_ImageAllocator.NotifyMediaType(&m_mtIn);
    return NOERROR;
}
// 输入 Pin 断开连接时，进行一些反初始化工作
HRESULT CVideoRenderer::BreakConnect()
{
    CAutoLock cInterfaceLock(&m_InterfaceLock);
    HRESULT hr = CBaseVideoRenderer::BreakConnect();
    if (FAILED(hr)) {
        return hr;
    }
    // The window is not used when disconnected
    IPin *pPin = m_InputPin.GetConnected();
    if (pPin)
        SendNotifyWindow(pPin,NULL);
    // 阻止重画视频窗口
    m_ImagePalette.RemovePalette();
    m_mtIn.ResetFormatBuffer();
    return NOERROR;
}
// 当输入 Pin 连接完成时，检查媒体类型描述的图像大小是否改变
// 如果改变，需要做其他一些相应设置
HRESULT CVideoRenderer::CompleteConnect(IPin *pReceivePin)
{
    CAutoLock cInterfaceLock(&m_InterfaceLock);
    CBaseVideoRenderer::CompleteConnect(pReceivePin);
    m_DrawImage.ResetPaletteVersion();
    // 判断图像大小是否改变（与 m_VideoSize 相比）
    VIDEOINFOHEADER *pVideoInfo = (VIDEOINFOHEADER *) m_mtIn.Format();
    if (pVideoInfo->bmiHeader.biWidth == m_VideoSize.cx)
    {
        if (pVideoInfo->bmiHeader.biHeight == m_VideoSize.cy)
        {
            return NOERROR;
        }
    }
```

```
    // 将视频窗口句柄传送给上一级 Filter 的输出 Pin（如果它支持 IMediaEventSink 接
    // 口的话）
    HWND hwnd = m_VideoText.GetWindowHWND();
    NOTE1("Sending EC_NOTIFY_WINDOW %x",hwnd);
    SendNotifyWindow(pReceivePin,hwnd);
    // 其他一些设置
    m_DrawImage.SetDrawContext();
    m_VideoSize.cx = pVideoInfo->bmiHeader.biWidth;
    m_VideoSize.cy = pVideoInfo->bmiHeader.biHeight;
    m_VideoText.SetDefaultSourceRect();
    m_VideoText.SetDefaultTargetRect();
    m_VideoText.OnVideoSizeChange();
    m_VideoText.ActivateWindow();
    return NOERROR;
}
```

另外还有 CVideoInputPin 类的两个函数：GetAllocator 和 NotifyAllocator。前者用于获得输入 Pin 上的 Sample 管理器（尽量使用 Filter 上创建的带有 DIBSECTION 数据结构的 Sample 管理器），后者用于通知 Pin 上实际使用的 Sample 管理器。

最后，来看一下 Filter 上暴露的接口：

```
STDMETHODIMP CVideoRenderer::NonDelegatingQueryInterface(REFIID
riid,void **ppv)
{
    CheckPointer(ppv,E_POINTER);
    if (riid == IID_ISpecifyPropertyPages) {
        return GetInterface((ISpecifyPropertyPages *)this, ppv);
    } else if (riid == IID_IVideoWindow) {
        return m_VideoText.NonDelegatingQueryInterface(riid,ppv);
    } else if (riid == IID_IBasicVideo) {
        return m_VideoText.NonDelegatingQueryInterface(riid,ppv);
    }
    return CBaseVideoRenderer::NonDelegatingQueryInterface(riid,ppv);
}
```

可以看到，ISpecifyPropertyPages 接口用于实现属性页，IVideoWindow 和 IbasicVideo 由 m_VideoText 对象来实现（这里牵涉到一个叫嵌套 COM 组件的概念），其他的接口（如 IMediaSeeking、IQualityControl、IQualProp 等）都由父类 CBaseVideoRenderer 来实现。

提示

Sample Video Renderer 的属性页实现，实际上就是 IQualProp 各接口方法的使用演示。这里不再展开。

14.2 基于 CBaseFilter 例子

这一节，将介绍另一个 Renderer Filter 的实现，它位于 SDK 安装目录 Samples\C++\DirectShow\Filters\Dump 下。

提示 与 DirectShow 标准的 File Writer（它的 CLSID 为 CLSID_FileWriter）相比，这个 Dump 例子实现的写文件功能的效率并不是很高。但 File Writer 只能接受 MEDIATYPE_Stream 类型的数据，而 Dump Filter 可以接受任何类型的数据。

14.2.1 实现的功能

Filter 的注册名："Dump"；

Filter 类型：Renderer Filter（仅有一个输入 Pin）；

CLSID：CLSID_Dump;

Merit 值：MERIT_DO_NOT_USE;

功能描述：接收任何类型的 Sample，并将 Sample 数据写入到指定的文件中。

运行如图 14.5 所示的 Filter Graph，就能将 AVI Decompressor 输出的各视频帧写到 FramesDump.dat 文件中。

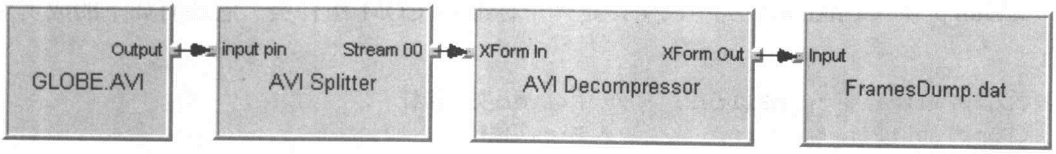

图 14.5 使用 Dump Filter 构建的 Filter Graph

14.2.2 实现要点

Filter 类名为 CDumpFilter，Pin 的类名为 CDumpInputPin。这些类的继承关系如图 14.6 所示。

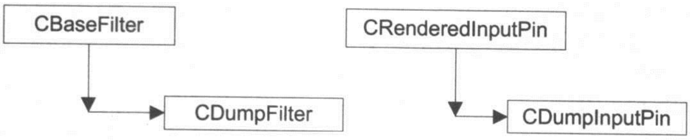

图 14.6 Dump Filter 的类继承关系

如图 14.6，这个 Renderer Filter 的父类选择了 CBaseFilter，这也是写 Renderer Filter 最常用的方法。因为 DirectShow 提供了标准的 Video Renderer 或 VMR，一般不再需要重复开发这种用于视频图像显示的 Filter，Renderer Filter 的父类也就很少会选择 CBaseVideoRenderer。另外，Renderer Filter 的输入 Pin 一般从 CRenderedInputPin 类中派生。可以在 SDK 基类源代码项目的 amextra.cpp 文件中找到 CRenderedInputPin 类的完整实现，它的主要功能是：当上一级 Filter 的所有数据都发送完毕，调用 Renderer Filter 的输入 Pin 上的 EndOfStream 函数后，这个 Pin 负责向 Filter Graph Manager 发送一个 EC_COMPLETE 事件。（一般当 Filter Graph 中的所有 Renderer Filter 都向 Filter Graph Manager 发送了一个 EC_COMPLETE 事件后，Filter Graph Manager 才会向应用程序发出一个 EC_COMPLETE 事件，表示整个 Filter Graph 播放完毕。）

除了上述两个类外，Dump 例子中还定义了另外一个 CDump 类，它的继承关系如图

14.7 所示。

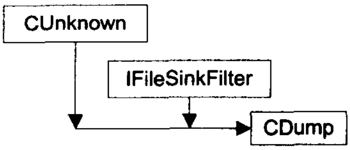

图 14.7　CDump 类的继承关系

再看 Filter 类工厂模版的定义：

```
CFactoryTemplate g_Templates[]= {
    L"Dump", &CLSID_Dump, CDump::CreateInstance, NULL, &sudDump
};
int g_cTemplates = 1;
```

可以看出，通过 CLSID_Dump 标识创建的 COM 组件并不是 CDumpFilter 类的实例，而是 CDump 类的实例。下面，不得不首先介绍一下这个嵌套的 COM 组件结构。

CDump 从 CUnknown 类中派生，是一个标准的 COM 组件类。它还有如下的成员变量：

```
CDumpFilter    *m_pFilter;    // Filter 实例指针
CDumpInputPin *m_pPin;        // Pin 实例指针
```

CDump 的构造函数直接创建了 Filter 和 Pin 的实例：

```
CDump::CDump(LPUNKNOWN pUnk, HRESULT *phr) :
    CUnknown(NAME("CDump"), pUnk),
    m_pFilter(NULL),
    m_pPin(NULL),
    m_pPosition(NULL),
    m_hFile(INVALID_HANDLE_VALUE),
    m_pFileName(0),
    m_fWriteError(0)
{
    ASSERT(phr);
    // 创建 Filter 实例
    m_pFilter = new CDumpFilter(this, GetOwner(), &m_Lock, phr);
    if (m_pFilter == NULL) {
        if (phr)
            *phr = E_OUTOFMEMORY;
        return;
    }
    // 创建 Pin 实例
    m_pPin = new CDumpInputPin(this,GetOwner(),
                               m_pFilter,
                               &m_Lock,
                               &m_ReceiveLock,
                               phr);
    if (m_pPin == NULL) {
        if (phr)
            *phr = E_OUTOFMEMORY;
```

```
            return;
        }
    }
```

已知 Filter 的基本特性主要体现在 IBaseFilter、IMediaFilter 等接口上，但通过 CLSID_Dump 标识创建的是 CDump 类的实例，又如何在 CDump 类上获得上述这些接口呢？可以来看一下 CDump::NonDelegatingQueryInterface 函数的实现，如下：

```
STDMETHODIMP CDump::NonDelegatingQueryInterface(REFIID riid, void ** ppv)
{
    CheckPointer(ppv,E_POINTER);
    CAutoLock lock(&m_Lock);
    // Do we have this interface
    if (riid == IID_IFileSinkFilter) {
        return GetInterface((IFileSinkFilter *) this, ppv);
    }
    else if (riid == IID_IBaseFilter || riid == IID_IMediaFilter || riid
            == IID_IPersist) {
        return m_pFilter->NonDelegatingQueryInterface(riid, ppv);
    }
    else if (riid == IID_IMediaPosition || riid == IID_IMediaSeeking) {
        if (m_pPosition == NULL)
        {
            HRESULT hr = S_OK;
            m_pPosition = new CPosPassThru(NAME("Dump Pass Through"),
                                           (IUnknown *) GetOwner(),
                                           (HRESULT *) &hr, m_pPin);
            if (m_pPosition == NULL)
                return E_OUTOFMEMORY;
            if (FAILED(hr))
            {
                delete m_pPosition;
                m_pPosition = NULL;
                return hr;
            }
        }
        return m_pPosition->NonDelegatingQueryInterface(riid, ppv);
    }
    return CUnknown::NonDelegatingQueryInterface(riid, ppv);
}
```

实际上，IBaseFilter、IMediaFilter 等接口还是“委托”给了 CDumpFilter 类对象去实现。这种 Filter 实现的嵌套结构，有点类似于 COM 的“聚合”原理。

接下来，分析一下这个 Filter 的具体实现。可以看到，CDumpFilter 类重新实现了父类的 GetPinCount 和 GetPin 函数，用于表明 Filter 拥有的 Pin 的数量以及根据 Pin 的索引号得到 Pin 对象指针：

```
// 这个 Filter 仅有一个 Pin
int CDumpFilter::GetPinCount()
{
```

```
    return 1;
}
CBasePin * CDumpFilter::GetPin(int n)
{
    if (n == 0) {
        return m_pDump->m_pPin;
    } else {
        return NULL;
    }
}
```

另外，还有 Run、Pause、Stop 三个函数的实现，用于打开或者关闭指定的文件：

```
STDMETHODIMP CDumpFilter::Run(REFERENCE_TIME tStart)
{
    CAutoLock cObjectLock(m_pLock);
    // Clear the global 'write error' flag that would be set
    // if we had encountered a problem writing the previous dump file.
    // (eg. running out of disk space).
    //
    // Since we are restarting the graph, a new file will be created.
    m_pDump->m_fWriteError = FALSE;
    // 打开文件
    if (m_pDump)
        m_pDump->OpenFile();
    return CBaseFilter::Run(tStart);
}
STDMETHODIMP CDumpFilter::Pause()
{
    CAutoLock cObjectLock(m_pLock);
    if (m_pDump)
    {
        // GraphEdit calls Pause() before calling Stop() for this filter.
        // If we have encountered a write error (such as disk full),
        // then stopping the graph could cause our log to be deleted
        // (because the current log file handle would be invalid).
        //
        // To preserve the log, don't open/create the log file on pause
        // if we have previously encountered an error.  The write error
        // flag gets cleared when setting a new log file name or
        // when restarting the graph with Run().
        if (!m_pDump->m_fWriteError)
        {
            m_pDump->OpenFile();
        }
    }
    return CBaseFilter::Pause();
}
STDMETHODIMP CDumpFilter::Stop()
{
    CAutoLock cObjectLock(m_pLock);
```

```
    // 关闭文件
    if (m_pDump)
        m_pDump->CloseFile();
    return CBaseFilter::Stop();
}
```

再看 CDumpInputPin 类的实现：

```
// 可以接受任何类型的数据输入
HRESULT CDumpInputPin::CheckMediaType(const CMediaType *)
{
    return S_OK;
}
// 表明不会再阻塞数据线程
STDMETHODIMP CDumpInputPin::ReceiveCanBlock()
{
    return S_FALSE;
}
// 每次接收到 Sample 后，将 Sample 的数据写入到指定的文件中
STDMETHODIMP CDumpInputPin::Receive(IMediaSample *pSample)
{
    CheckPointer(pSample,E_POINTER);
    CAutoLock lock(m_pReceiveLock);
    PBYTE pbData;
    // Has the filter been stopped yet?
    if (m_pDump->m_hFile == INVALID_HANDLE_VALUE) {
        return NOERROR;
    }
    REFERENCE_TIME tStart, tStop;
    pSample->GetTime(&tStart, &tStop);
    DbgLog((LOG_TRACE, 1, TEXT("tStart(%s), tStop(%s), Diff(%d ms),
            Bytes(%d)"),
            (LPCTSTR) CDisp(tStart),
            (LPCTSTR) CDisp(tStop),
            (LONG)((tStart - m_tLast) / 10000),
            pSample->GetActualDataLength()));
    m_tLast = tStart;
    // Copy the data to the file
    HRESULT hr = pSample->GetPointer(&pbData);
    if (FAILED(hr)) {
        return hr;
    }
    return m_pDump->Write(pbData, pSample->GetActualDataLength());
}
```

最后，来看一下 Filter 上的接口实现情况。已知 Renderer Filter 应该在 Filter 上暴露 IMediaSeeking 或 IMediaPosition 接口，并负责把各种随机定位操作通过（连接着的）输入 Pin 往上一级 Filter 发送，如 CDump::NonDelegatingQueryInterface 函数中，使用 CposPass-Thru 类来辅助实现这一功能。值得注意的是，当输入 Pin 断开的时候，需要调用 CPosPassThru::ForceRefresh 函数，实现如下：

```
HRESULT CDumpInputPin::BreakConnect()
{
    if (m_pDump->m_pPosition != NULL) {
        m_pDump->m_pPosition->ForceRefresh();
    }
    return CRenderedInputPin::BreakConnect();
}
```

另外，当在 GraphEdit 中加入一个 Dump Filter 时，会自动弹出一个对话框，要求指定一个数据输出文件名。这主要依赖于 IFileSinkFilter 接口的实现。可以在 CDump 类中看到 IFileSinkFilter 的两个接口方法：SetFileName 和 GetCurFile 的实现，如下：

```
STDMETHODIMP CDump::SetFileName(LPCOLESTR pszFileName,const
AM_MEDIA_TYPE *pmt)
{
    // 文件名合法性检查
    CheckPointer(pszFileName,E_POINTER);
    if(wcslen(pszFileName) > MAX_PATH)
        return ERROR_FILENAME_EXCED_RANGE;
    // 保存文件名
    m_pFileName = new WCHAR[1+lstrlenW(pszFileName)];
    if (m_pFileName == 0)
        return E_OUTOFMEMORY;
    lstrcpyW(m_pFileName,pszFileName);
    // Clear the global 'write error' flag that would be set
    // if we had encountered a problem writing the previous dump file.
    // (eg. running out of disk space).
    m_fWriteError = FALSE;
    //打开并随后关闭该文件，判断文件路径的有效性
    HRESULT hr = OpenFile();
    CloseFile();
    return hr;
}
STDMETHODIMP CDump::GetCurFile(LPOLESTR * ppszFileName,AM_MEDIA_TYPE
                                *pmt)
{
    CheckPointer(ppszFileName, E_POINTER);
    *ppszFileName = NULL;
    // 将文件名拷贝出去
    if (m_pFileName != NULL)
    {
        *ppszFileName = (LPOLESTR)
        QzTaskMemAlloc(sizeof(WCHAR) * (1+lstrlenW(m_pFileName)));
        if (*ppszFileName != NULL)
        {
            lstrcpyW(*ppszFileName, m_pFileName);
        }
    }
    // 指定一种媒体类型
    if(pmt)
```

```
    {
        ZeroMemory(pmt, sizeof(*pmt));
        pmt->majortype = MEDIATYPE_NULL;
        pmt->subtype = MEDIASUBTYPE_NULL;
    }
    return S_OK;
}
```

第 15 章　DMO 例 子

15.1　实现的功能

本章主要介绍一个 DMO 的例子，它位于 SDK 安装目录 Samples\C++\DirectShow\DMO\DMOSample 下。

这个 DMO 的注册名为“DMO Sample”，CLSID 为 CLSID_Sample，有一个输入流、两个输出流。它完成的主要工作是：接收 MPEG1 的压缩视频数据，然后输出纯色背景的叠加有时间戳的非压缩图像帧。可以知道，DMO 通过 DMO Wrapper Filter 参与到 Filter Graph 中，如图 15.1 所示。

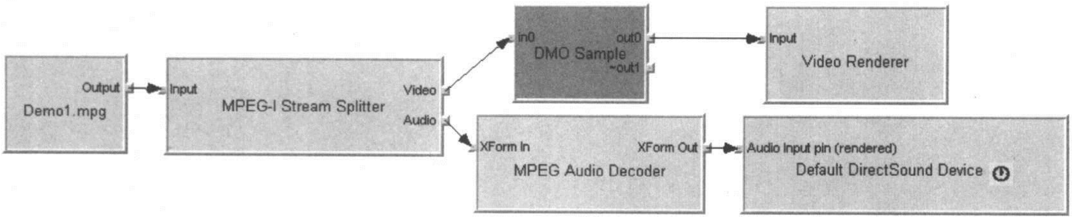

图 15.1　使用 DMO Sample 构建的 Filter Graph

运行这个 Filter Graph，可以看到如图 15.2 所示的视频窗口输出。

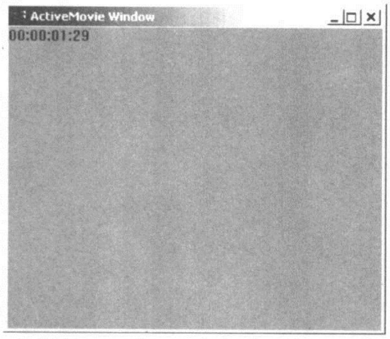

图 15.2　DMO Sample 的输出

15.2　实 现 要 点

已知 DMO 是一种实现了 IMediaObject 接口的 COM 组件；一般可以使用一个 ATL 项目来实现。下面，来看一下组件实现的框架。

首先，实现这个 DMO 的 DLL 要导出 4 个函数（在 dmosample.def 中声明）：

```
LIBRARY     "dmosample.DLL"
EXPORTS
            DllCanUnloadNow      PRIVATE
            DllGetClassObject    PRIVATE
            DllRegisterServer    PRIVATE
            DllUnregisterServer  PRIVATE
```

在 dmosample.cpp 文件中实现如下：

```
// Used to determine whether the DLL can be unloaded by OLE
STDAPI DllCanUnloadNow(void)
{
    return (_Module.GetLockCount()==0) ? S_OK : S_FALSE;
}
// Returns a class factory to create an object of the requested type
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    return _Module.GetClassObject(rclsid, riid, ppv);
}
// DllRegisterServer - Adds entries to the system registry
STDAPI DllRegisterServer(void)
{
    // Register ourselves as a DMO with no types
    // RGS file also registers with a merit of 0
    //so DirectShow won't try to use us
    DMORegister(L"DMO Sample",
                CLSID_Sample,
                DMOCATEGORY_VIDEO_DECODER,
                0,
                0,
                NULL,
                0,
                NULL);
    // registers object
    return _Module.RegisterServer();
}
// DllUnregisterServer - Removes entries from the system registry
STDAPI DllUnregisterServer(void)
{
    DMOUnregister(CLSID_Sample, DMOCATEGORY_VIDEO_DECODER);
    return _Module.UnregisterServer();
}
```

下面再来看组件类 CSample 的实现。CSample 类的定义如下：

```
class ATL_NO_VTABLE CSample :
    public IMediaObjectImpl<CSample, 1, 2>,        // 1 input, 2 outputs
        public CComObjectRootEx<CComMultiThreadModel>,
        public CComCoClass<CSample, &CLSID_Sample>
```

这个 DMO 使用了模板类 IMediaObjectImpl 来辅助实现，有一个输入流、两个输出流。于是，需要实现 IMediaObjectImpl 类的一系列以“Internal”开头的回调函数。

输入流的类型没有特殊要求，而输出流 0 要求每次输出单帧的非压缩图像，输出流 1 指定为可选流，参见 InternalGetInputStreamInfo 和 InternalGetOutputStreamInfo 的实现：

```
HRESULT CSample::InternalGetInputStreamInfo(DWORD dwInputStreamIndex,
DWORD *pdwFlags)
{
    //  We can process data on any boundary
    *pdwFlags = 0;
    return S_OK;
}
HRESULT CSample::InternalGetOutputStreamInfo(DWORD dwOutputStreamIndex,
                                             DWORD *pdwFlags)
{
    //  We output single frames
    if (0 == dwOutputStreamIndex) {
        *pdwFlags = DMO_OUTPUT_STREAMF_WHOLE_SAMPLES |
                    DMO_OUTPUT_STREAMF_SINGLE_SAMPLE_PER_BUFFER |
                    DMO_OUTPUT_STREAMF_FIXED_SAMPLE_SIZE;
    } else {
        //  Stream 1
        //  Just text, no special buffering but 1 sample per sample
        *pdwFlags = DMO_OUTPUT_STREAMF_SINGLE_SAMPLE_PER_BUFFER |
                    DMO_OUTPUT_STREAMF_OPTIONAL;
    }
    return S_OK;
}
```

输入、输出流上媒体类型的检查，实现如下：

```
// 输入流上仅接受 MEDIASUBTYPE_MPEG1Payload 的媒体类型
HRESULT CSample::InternalCheckInputType(DWORD dwInputStreamIndex, const
                                        DMO_MEDIA_TYPE *pmt)
{
    //  判断当前流上是否已经设置了媒体类型
    if (InputTypeSet(dwInputStreamIndex)) {
        if (!TypesMatch(pmt, InputType(dwInputStreamIndex))) {
            return DMO_E_INVALIDTYPE;
        } else {
            return S_OK;
        }
    }
    //  仅接受 MEDIATYPE_Video, MEDIASUBTYPE_MPEG1Video 的媒体类型
    //  同时检查是否定义了格式块
    if (pmt->majortype == MEDIATYPE_Video &&
        pmt->subtype == MEDIASUBTYPE_MPEG1Payload &&
        pmt->formattype == FORMAT_MPEGVideo &&
        pmt->pbFormat != NULL) {
        return S_OK;
    } else {
        return DMO_E_INVALIDTYPE;
    }
```

```
}
HRESULT CSample::InternalCheckOutputType(DWORD dwOutputStreamIndex,
                                          const DMO_MEDIA_TYPE *pmt)
{
    //  判断当前流上是否已经设置了媒体类型
    if (OutputTypeSet(dwOutputStreamIndex)) {
        if (!TypesMatch(pmt, OutputType(dwOutputStreamIndex))) {
            return DMO_E_INVALIDTYPE;
        } else {
            return S_OK;
        }
    }
    //  在输出流 0 上输出 RGB565 格式的图像帧
    //  要求输入流先设置媒体类型
    if (dwOutputStreamIndex == 0) {
        if (!InputTypeSet(0)) {
            return DMO_E_INVALIDTYPE;
        }
        if (pmt->majortype == MEDIATYPE_Video &&
            pmt->subtype == MEDIASUBTYPE_RGB565 &&
            pmt->formattype == FORMAT_VideoInfo &&
            pmt->pbFormat != NULL) {
            const VIDEOINFOHEADER *pvihInput = (const VIDEOINFOHEADER
                                                *)InputType(0)->pbFormat;
            const VIDEOINFOHEADER *pvihOutput = (const VIDEOINFOHEADER
                                                 *)pmt->pbFormat;
            LONG lWidth, lHeight;
            if (IsRectEmpty(&pvihOutput->rcTarget)) {
                lWidth = pvihOutput->bmiHeader.biWidth;
                lHeight = pvihOutput->bmiHeader.biHeight;
            } else {
                lWidth = pvihOutput->rcTarget.right - pvihOutput->
                         rcTarget.left;
                lHeight = pvihOutput->rcTarget.bottom - pvihOutput->
                          rcTarget.top;
            }
            //  输入、输出图像大小必须保持一致
            if (pvihInput->bmiHeader.biWidth == lWidth &&
                pvihInput->bmiHeader.biHeight == lHeight) {
                return S_OK;
            }
        }
        return DMO_E_INVALIDTYPE;
    } else {
        if (pmt->majortype == MEDIATYPE_Text && pmt->subtype ==
            GUID_NULL) {
            return S_OK;
        } else {
            return DMO_E_INVALIDTYPE;
        }
```

```
    }
}
```

DMOSample 例子仅在输出流上提供首选媒体类型：

```
HRESULT CSample::InternalGetInputType(DWORD dwInputStreamIndex, DWORD
                                     dwTypeIndex,DMO_MEDIA_TYPE *pmt)
{
    //  在输入流上不提供首选媒体类型
    return DMO_E_NO_MORE_ITEMS;
}
HRESULT CSample::InternalGetOutputType(DWORD dwOutputStreamIndex, DWORD
                                     dwTypeIndex,DMO_MEDIA_TYPE *pmt)
{
    //  要求输入流先设置媒体类型
    if (!InputTypeSet(0)) {
        return DMO_E_TYPE_NOT_SET;
    }
    // 仅提供一种首选媒体类型
    if (dwTypeIndex != 0) {
        return DMO_E_NO_MORE_ITEMS;
    }
    // If GetOutputType()'s pmt parameter is NULL, return S_OK if the
    // type exists.Return DMO_E_NO_MORE_ITEMS if the type does not
    // exists. See the documentation for IMediaObject::GetOutputType()
    // for more information.
    if (NULL != pmt) {
        if (dwOutputStreamIndex == 0) {
            //  Create our media type
            HRESULT hr = MoInitMediaType(pmt, FIELD_OFFSET(VIDEOINFO,
                        dwBitMasks[3]));
            if (FAILED(hr)) {
                return hr;
            }
            const VIDEOINFOHEADER *pvihInput = (const VIDEOINFOHEADER
                                    *)InputType(0)->pbFormat;
            LONG lWidth  = pvihInput->bmiHeader.biWidth;
            LONG lHeight = pvihInput->bmiHeader.biHeight;
            //  Initialize the media type structure (MoInitMediaType
            //initalized cbFormat and pbFormat)
            pmt->majortype             = MEDIATYPE_Video;
            pmt->subtype               = MEDIASUBTYPE_RGB565;
            pmt->bFixedSizeSamples     = TRUE;
            pmt->bTemporalCompression  = FALSE;
            pmt->lSampleSize           = lWidth * lHeight * 2;
            pmt->formattype            = FORMAT_VideoInfo;
            pmt->pUnk                  = NULL;
            //  Initialize the format
            VIDEOINFO *pviOutput = (VIDEOINFO *)pmt->pbFormat;
            ZeroMemory(pviOutput, FIELD_OFFSET(VIDEOINFO,
                    dwBitMasks[3]));
```

```
            pviOutput->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
            pviOutput->bmiHeader.biCompression = BI_BITFIELDS;
            pviOutput->bmiHeader.biBitCount = 16;
            pviOutput->bmiHeader.biPlanes = 1;
            pviOutput->bmiHeader.biWidth = lWidth;
            pviOutput->bmiHeader.biHeight = lHeight;
            pviOutput->bmiHeader.biSizeImage = pmt->lSampleSize;
            pviOutput->TrueColorInfo.dwBitMasks[0] = 0xF800;
            pviOutput->TrueColorInfo.dwBitMasks[1] = 0x07E0;
            pviOutput->TrueColorInfo.dwBitMasks[2] = 0x001F;
            pviOutput->AvgTimePerFrame = pvihInput->AvgTimePerFrame;
        } else {
            ZeroMemory(pmt, sizeof(*pmt));
            pmt->majortype = MEDIATYPE_Text;
        }
    }
    return S_OK;
}
```

最后，重点看一下实际的数据处理过程。因为 DMOSample 接收的是 MPEG1 的压缩视频数据，那又怎么来输出非压缩的图像帧呢？DMOSample 的解决办法是：分析 MPEG1 数据，找出图像头标识（0x00000100），每找到一个这样的标识，就让输出流输出一个图像帧。可以看到 CSample 类中定义了一个布尔变量 m_bPicture，就是来表示是否可以输出图像帧的。可以参考 InternalProcessInput 和 InternalProcessOutput 函数的实现，如下：

```
HRESULT CSample::InternalProcessInput(DWORD dwInputStreamIndex,
IMediaBuffer *pBuffer,DWORD dwFlags, REFERENCE_TIME rtTimestamp,
REFERENCE_TIME rtTimelength)
{
    //  Check parameters
    _ASSERTE(m_pBuffer == NULL);
    //  取得数据指针和数据长度
    HRESULT hr = pBuffer->GetBufferAndLength(&m_pbData, &m_cbData);
    if (FAILED(hr)) {
        return hr;
    }
    m_pBuffer = pBuffer;  // 保存当前的数据缓存对象指针
    if (0 == (dwFlags & DMO_INPUT_DATA_BUFFERF_TIME)) {
        rtTimestamp = INVALID_TIME;
    }
    m_StreamState.TimeStamp(rtTimestamp);

    //  Process() returns S_FALSE if there is no output, S_OK otherwise
    hr = Process();   // 处理输入的数据，寻找图像头标识，会影响m_bPicture 标记
    return hr;
}
HRESULT CSample::InternalProcessOutput(DWORD dwFlags,
                                       DWORD cOutputBufferCount,
                                       DMO_OUTPUT_DATA_BUFFER
                                       *pOutputBuffers,
```

```
                                          DWORD *pdwStatus)
{
    //  Check buffer
    PBYTE pbData;
    DWORD cbData;
    DWORD cbCurrent;
    //  判断是否可以输出图像帧，m_bPicture 在 InternalProcessInput 中设置
    if (!m_bPicture) {
        return S_FALSE;
    }
    HRESULT hr = pOutputBuffers[0].pBuffer->GetBufferAndLength(&pbData,
                                                        &cbCurrent);
    if (FAILED(hr)) {
        return hr;
    }
    hr = pOutputBuffers[0].pBuffer->GetMaxLength(&cbData);
    if (FAILED(hr)) {
        return hr;
    }
    if (cbData < cbCurrent + (DWORD)OutputType(0)->lSampleSize) {
        return E_INVALIDARG;
    }
    //  Say we've filled the buffer
    hr = pOutputBuffers[0].pBuffer->SetLength(cbCurrent +
        (DWORD)OutputType(0)->lSampleSize);
    cbData -= cbCurrent;
    pbData += cbCurrent;
    //  产生时间戳
    DWORD dwTimeCode;
    REFERENCE_TIME rt = m_StreamState.PictureTime(&dwTimeCode);
    TCHAR szBuffer[20];
    //  生成时间戳字符串
    wsprintf(szBuffer, TEXT("%2.2d:%2.2d:%2.2d:%2.2d\0"),
             (dwTimeCode >> 19) & 0x1F,
             (dwTimeCode >> 13) & 0x3F,
             (dwTimeCode >> 6) & 0x3F,
              dwTimeCode & 0x3F);
    //  用纯色刷新图像帧
    OurFillRect((const VIDEOINFOHEADER *)OutputType(0)->pbFormat, pbData,
                0x03EF);
    //  叠加时间戳到图像帧上
    DrawOurText((const VIDEOINFOHEADER *)OutputType(0)->pbFormat, pbData,
                szBuffer);
    //  非压缩的图像帧都可以作为同步点
    pOutputBuffers[0].dwStatus = DMO_OUTPUT_DATA_BUFFERF_SYNCPOINT;
    //  设置时间戳
    if (rt != INVALID_TIME) {
        pOutputBuffers[0].rtTimestamp = rt;
    } else {
        pOutputBuffers[0].rtTimestamp = m_rtFrame;
```

```
    }
    REFERENCE_TIME rtLength = ((const VIDEOINFOHEADER *)OutputType(0)->
                               pbFormat)->AvgTimePerFrame;
    pOutputBuffers[0].rtTimelength = rtLength;
    m_rtFrame = pOutputBuffers[0].rtTimestamp + rtLength;
    //  Uncompressed video must always have a timestamp
    pOutputBuffers[0].dwStatus |= DMO_OUTPUT_DATA_BUFFERF_TIME |
                                  DMO_OUTPUT_DATA_BUFFERF_TIMELENGTH;
    //  Update our state
    m_bPicture = false;
    //  再次调用 Process 函数处理数据，如果又找到一个图像头，则在输出的数据缓存对象
    //  上打上 DMO_OUTPUT_DATA_BUFFERF_INCOMPLETE 标记
    if (S_OK == Process()) {
        pOutputBuffers[0].dwStatus = DMO_OUTPUT_DATA_BUFFERF_INCOMPLETE;
    }
    //  判断是否需要输出可选流
    if (pOutputBuffers[1].pBuffer) {
        PBYTE pbData;
        DWORD cbLength;
        DWORD cbMax;
        DWORD dwLen = lstrlen(szBuffer) * sizeof(WCHAR);
        if (S_OK == pOutputBuffers[1].pBuffer->
                    GetBufferAndLength(&pbData, &cbLength) &&
            S_OK == pOutputBuffers[1].pBuffer->GetMaxLength(&cbMax) &&
                    cbLength + dwLen <= cbMax) {
            // Convert to UNICODE
            USES_CONVERSION;
            LPWSTR lpsz = T2W(szBuffer);
            CopyMemory(pbData + cbLength, lpsz, dwLen);
            pOutputBuffers[1].pBuffer->SetLength(cbLength + dwLen);
            pOutputBuffers[1].dwStatus = pOutputBuffers[0].dwStatus;
            pOutputBuffers[1].rtTimestamp = pOutputBuffers[0].
                                            rtTimestamp;
            pOutputBuffers[1].rtTimelength = rtLength;
        }
    }
    return S_OK;
}
// MPEG1 压缩视频数据的分析过程
HRESULT CSample::Process()
{
    //  输入数据全部分析完，或者找到了一个图像头后跳出 while 循环
    while (m_cbData && !m_bPicture) {
        m_bPicture = m_StreamState.NextByte(*m_pbData);
        m_cbData--;
        m_pbData++;
    }
    //  如果输入数据都被分析完了，m_pBuffer 指针赋值为 NULL
    if (m_cbData == 0) {
        m_pBuffer = NULL;
```

```
    }
    // assert that if have no picture to output then we ate all the data
    _ASSERTE(m_bPicture || m_cbData == 0);
    return m_bPicture ? S_OK : S_FALSE;
}
```

提示 上述函数中，OurFillRect、DrawOurText、TypesMatch 等均是在 util.cpp 文件中实现的工具函数。MPEG1 视频数据的具体分析过程，参见 state.cpp 文件中 CStreamState::NextByte 函数的实现。MPEG1 流的时间戳获得位置在紧跟图组头 GOP（0x000001b8）后面的 25 位数据。

第 16 章　枚 举 例 子

16.1　系统枚举例子

这一节，我们将介绍一个系统枚举例子的实现，它位于 SDK 安装目录 Samples\C++\DirectShow\Misc\SysEnum 下。

16.1.1　实现的功能

这是一个 Filter 的枚举程序，类似于 GraphEdit 执行菜单项 Graph | Insert Filters...后弹出的选择 Filter 的对话框。运行 SysEnum 程序，如图 16.1 所示。

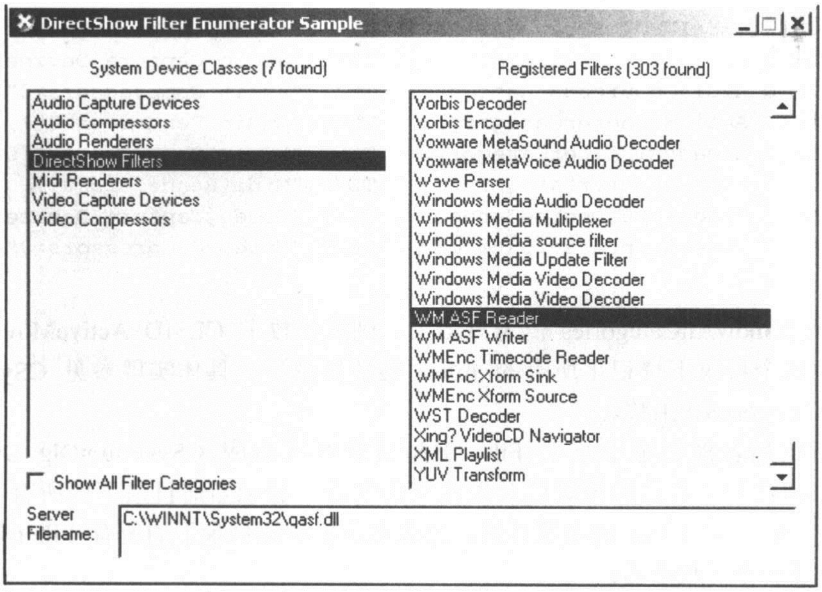

图 16.1　SysEnum 程序的界面

在图 16.1 中，左边列表框中为 Filter 类型目录的名称，右边列表框中为当前 Filter 类型目录下的所有 Filter。另外，选中 Show All Filter Categories 复选框可以在左边列表框中显示所有的 Filter 类型目录名；Server Filename 文本框中可以显示当前选中的 Filter 的实现文件路径。

16.1.2　实现要点

Filter 的枚举依赖于系统枚举器——DirectShow 提供的一个标识为 CLSID_SystemDevice-Enum 的 COM 组件。可以看到，SysEnum 是一个基于对话框的 MFC 程序；下面在主对话框的初始化函数 CSysEnumDlg::OnInitDialog 中创建这个枚举器，如下：

```
CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);
// Instantiate the system device enumerator
m_pSysDevEnum = NULL;
hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL,
                      CLSCTX_INPROC, IID_ICreateDevEnum,
                      (void **)&m_pSysDevEnum);
if FAILED(hr)
{
    CoUninitialize();
    return FALSE;
}
```

接下来填写左边的列表框（即 Filter 的类型目录名）。不难发现，CSysEnumDlg 类定义了一个成员变量 m_bShowAllCategories，用于指示 Show All Filter Categories 是否处于选中状态。如果 m_bShowAllCategories 值为 FALSE，则填入应用程序上定义的常见的 7 个目录：

```
const CATEGORY_INFO_DS categories[] = {
    &CLSID_AudioInputDeviceCategory, TEXT("Audio Capture Devices\0"),
    &CLSID_AudioCompressorCategory,  TEXT("Audio Compressors\0"),
    &CLSID_AudioRendererCategory,    TEXT("Audio Renderers\0"),
    &CLSID_LegacyAmFilterCategory,   TEXT("DirectShow Filters\0"),
    &CLSID_MidiRendererCategory,     TEXT("Midi Renderers\0"),
    &CLSID_VideoInputDeviceCategory, TEXT("Video Capture Devices\0"),
    &CLSID_VideoCompressorCategory,  TEXT("Video Compressors\0"),
};
```

如果 m_bShowAllCategories 值为 TRUE，则需要枚举 CLSID_ActiveMovieCategories 目录（因为这个目录下登记了所有的 Filter 类型目录）。具体实现参见 CSysEnumDlg::DisplayFullCategorySet 函数。

当在左边列表框中选中一个 Filte 类型目录时（响应 CSysEnumDlg::OnSelchange-ListDevices 函数），右边的列表框会做相应的刷新，显示当前目录下的所有 Filter 的名字。那么，给定一个 Filter 的类型目录，怎么实现枚举该目录下的所有 Filter 呢？首先要为该目录创建一个子枚举器：

```
hr = m_pSysDevEnum->CreateClassEnumerator(*clsid, &pEnumCat, 0);
```

然后再用这个子枚举器枚举指定的目录，参见 CSysEnumDlg::EnumFilters 函数的实现：

```
HRESULT CSysEnumDlg::EnumFilters(IEnumMoniker *pEnumCat)
{
    HRESULT hr=S_OK;
    IMoniker *pMoniker;
    ULONG cFetched;
    VARIANT varName={0};
    int nFilters=0;
    // 清除当前 Filter 列表框中的所有内容
    ClearFilterList();
```

```
// If there are no filters of a requested type, show default string
if (!pEnumCat)
{
    m_FilterList.AddString(TEXT("<< No entries >>"));
    SetNumFilters(nFilters);
    return S_FALSE;
}
// 枚举当前目录下的所有 Filter
while(pEnumCat->Next(1, &pMoniker, &cFetched) == S_OK)
{
    IPropertyBag *pPropBag;
    ASSERT(pMoniker);
    // 将标识连接到一个文件
    hr = pMoniker->BindToStorage(0, 0, IID_IPropertyBag,
                                 (void **)&pPropBag);
    ASSERT(SUCCEEDED(hr));
    ASSERT(pPropBag);
    if (FAILED(hr))
        continue;
    // 取得 Filter 的名字（Friendly Name）
    varName.vt = VT_BSTR;
    hr = pPropBag->Read(L"FriendlyName", &varName, 0);
    if (FAILED(hr))
        continue;
    // Get filter name (converting BSTR name to a CString)
    CString str(varName.bstrVal);
    SysFreeString(varName.bstrVal);
    nFilters++;
    // 获得 Filter 的 CLSID 字符串
    VARIANT varFilterClsid;
    varFilterClsid.vt = VT_BSTR;
    // Read CLSID string from property bag
    hr = pPropBag->Read(L"CLSID", &varFilterClsid, 0);
    if(SUCCEEDED(hr))
    {
        CLSID clsidFilter;
        // 将字符串 CLSID 转化为二进制变量后加入到列表框中
        if(CLSIDFromString(varFilterClsid.bstrVal, &clsidFilter) ==
           S_OK)
        {
            AddFilter(str, &clsidFilter);
        }
        SysFreeString(varFilterClsid.bstrVal);
    }
    // Cleanup interfaces
    SAFE_RELEASE(pPropBag);
    SAFE_RELEASE(pMoniker);
}
// Update count of enumerated filters
SetNumFilters(nFilters);
```

```
    return hr;
}
```

提示

在 Debug 模式下枚举 Filter，有时候会出现异常（int 3 中断）。那是因为有些第三方 Filter 为了防止反编译，加入了一种保护措施。不过，在 Release 版本下执行程序不会有这种问题。

然后，当在右边的列表框中选中一个 Filter 时（响应 CSysEnumDlg::OnSelchangeList-Filters 函数），需要在 Server Filename 文本框中显示 Filter 的实现文件的路径。参见 CSysEnumDlg::ShowFilenameByCLSID 函数的实现：

```
void CSysEnumDlg::ShowFilenameByCLSID(REFCLSID clsid)
{
    HRESULT hr;
    LPOLESTR strCLSID;
    // 将二进制变量形式的 CLSID 转化为字符串形式
    hr = StringFromCLSID(clsid, &strCLSID);
    if(SUCCEEDED(hr))
    {
        TCHAR szKey[512];
        CString strQuery(strCLSID);
        // 创建注册表中的子键名
        wsprintf(szKey, TEXT("Software\\Classes\\CLSID\\%s\\
        InprocServer32\0"),
                (LPCTSTR) strQuery);
        // Free memory associated with strCLSID (allocated in
        //StringFromCLSID)
        CoTaskMemFree(strCLSID);
        HKEY hkeyFilter=0;
        DWORD dwSize=MAX_PATH;
        BYTE pbFilename[MAX_PATH];
        int rc=0;
        // 打开这个 Filter 的注册表子键
        rc = RegOpenKey(HKEY_LOCAL_MACHINE, szKey, &hkeyFilter);
        if (rc == ERROR_SUCCESS)
        {
            rc = RegQueryValueEx(hkeyFilter, NULL,//Read(Default) value
                                 NULL, NULL, pbFilename, &dwSize);
            if (rc == ERROR_SUCCESS)
            {
                TCHAR szFilename[MAX_PATH];
                wsprintf(szFilename, TEXT("%s\0"), pbFilename);
                // 将 Filter 的实现文件路径设置给 Server Filename 文本框
                m_StrFilename.SetWindowText(szFilename);
            }
            RegCloseKey(hkeyFilter);
        }
    }
}
```

最后，记得在程序退出前，释放系统枚举器对象，以及进行 COM 库的反初始化。

```
void CSysEnumDlg::OnClose()
{
    // Free any stored CLSID pointers (in listbox item data ptr area)
    ClearFilterList();
    ClearDeviceList();
    // Release system device enumerator and close COM
    SAFE_RELEASE(m_pSysDevEnum);
    CoUninitialize();
    CDialog::OnClose();
}
```

16.2　Filter 映射例子

这一节，将介绍一个 Filter 映射（查找）例子的实现，它位于 SDK 安装目录 Samples\C++\DirectShow\Misc\Mapper 下。

16.2.1　实现的功能

这是一个 Filter 的映射查找程序，比如指定了 Filter 的外部描述（如输入和输出 Pin 的数量、媒体类型，Filter 的 Merit 值等），可以得到系统中满足条件的所有 Filter。运行 Mapper 程序，如图 16.2 所示。

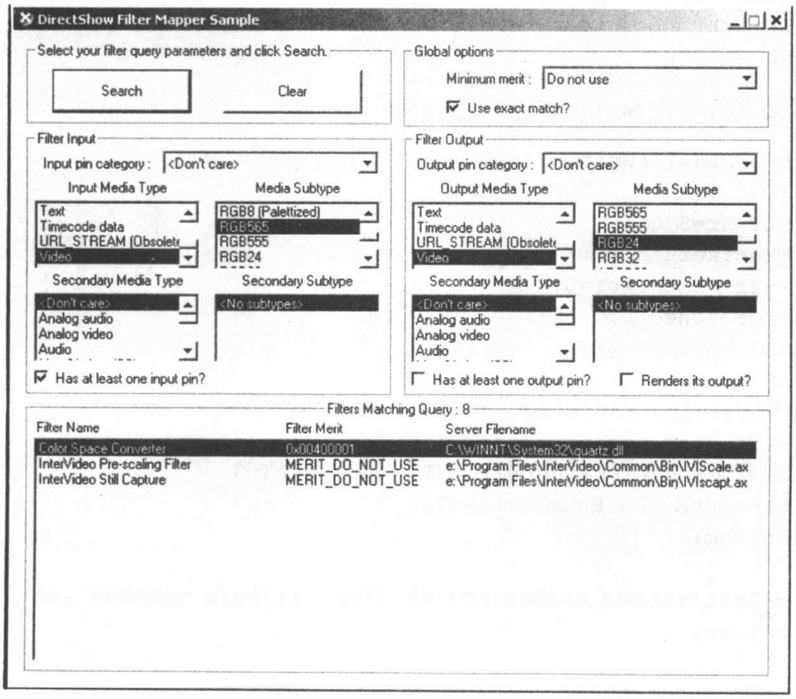

图 16.2　Mapper 程序的界面

在图 16.2 中，可以在 Filter Input 框内设置对 Filter 的输入 Pin 的描述，在 Filter Output 框内设置对 Filter 的输出 Pin 的描述。在 Global options 框内还可以设置 Filter 的最小 Merit 值（即 Filter 的 Merit 值必须大于或等于这个指定的值），以及设置是否进行严格的映射查找。当所有的描述条件都设置好了，就可以单击 Search 按钮进行查询；查询的结果将在 Filters Matching Query 框内显示出来。

例如，在 Filter Input 框内设置输入 Pin 的媒体类型为（Video，RGB565），在 Filter Output 框内设置输出 Pin 的媒体类型为（Video，RGB24），在 Global options 框中设置 Minimum merit 为 Do not use，并选中“Use exact match?”复选框，最后单击 Search 按钮。意料之中，查到了 DirectShow 提供的 Color Space Converter Filter。

16.2.2 实现要点

Filter 的映射查找依赖于 DirectShow 提供的一个标识为 CLSID_FilterMapper2 的 COM 组件（调用其接口方法 IFilterMapper2::EnumMatchingFilters）。可以看到，Mapper 例子是一个基于对话框的 MFC 程序；在主对话框的初始化函数 CMapperDlg::OnInitDialog 中创建这个映射器：

```
CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);
// Create the filter mapper that will be used for all queries
HRESULT hr = CoCreateInstance(CLSID_FilterMapper2,
                      NULL, CLSCTX_INPROC, IID_IFilterMapper2,
                      (void **) &m_pMapper);
ASSERT(SUCCEEDED(hr));
if (FAILED(hr))
    return FALSE;
```

下面，直接来看单击 Search 按钮时的响应：

```
void CMapperDlg::OnButtonSearch()
{
    HRESULT hr=S_OK;
    IEnumMoniker *pEnum;
    IMoniker *pMoniker=0;
    ULONG cFetched=0;
    VARIANT varName={0};
    int nFilters=0;
    // 清除 Filters Matching Query 列表框的内容
    m_ListFilters.ResetContent();
    // 根据界面上设置的 Filter 描述条件，得到一个枚举器
    pEnum = GetFilterEnumerator();
    if (!pEnum)
    {
        m_ListFilters.AddString(TEXT("<No filters matched query>"));
        return;
    }
    hr = pEnum->Reset();
    // 枚举符合条件的所有 Filter，并把它们加入列表框中
    while((hr = pEnum->Next(1, &pMoniker, &cFetched)) == S_OK)
```

```
{
    IPropertyBag *pPropBag;
    ASSERT(pMoniker);
    // 将标识连接到一个文件
    hr = pMoniker->BindToStorage(0, 0, IID_IPropertyBag,
                                 (void **)&pPropBag);
    ASSERT(SUCCEEDED(hr));
    ASSERT(pPropBag);
    if (FAILED(hr))
        continue;
    // 获得 Filter 的 Friendly Name
    varName.vt = VT_BSTR;
    hr = pPropBag->Read(L"FriendlyName", &varName, 0);
    if (FAILED(hr))
        continue;
    // Get filter name (converting BSTR name to a CString)
    CString str(varName.bstrVal);
    SysFreeString(varName.bstrVal);
    nFilters++;
    // 获得 Filter 的 Merit 值
    DWORD dwMerit;
    GetMerit(pPropBag, &dwMerit);
    // Read filter's CLSID from property bag.  This CLSID string
    // will be converted to a binary CLSID and passed to AddFilter(),
    // which will add the filter's name to the listbox and its CLSID
    // to the listbox item's DataPtr item.
    VARIANT varFilterClsid;
    varFilterClsid.vt = VT_BSTR;
    // 获得 Filter 的 CLSID 字符串
    hr = pPropBag->Read(L"CLSID", &varFilterClsid, 0);
    if(SUCCEEDED(hr))
    {
        CLSID clsidFilter;
        // Add filter name and filename to listbox
        if(CLSIDFromString(varFilterClsid.bstrVal, &clsidFilter) ==
        S_OK)
        {
            AddFilter(str, &clsidFilter, dwMerit);
        }
        SysFreeString(varFilterClsid.bstrVal);
    }
    // Clean up interfaces
    SAFE_RELEASE(pPropBag);
    SAFE_RELEASE(pMoniker);
}
// Display number of filters matching query
TCHAR szNumFilters[48];
wsprintf(szNumFilters, TEXT("Filters Matching Query : %d\0"),
         nFilters);
m_StrNumFilters.SetWindowText(szNumFilters);
```

```
    // Clean up enumerator
    if (pEnum)
        pEnum->Release();
}
```

可以看到，OnButtonSearch 函数中用到了几个重要的工具函数，它们是：GetFilter-Enumerator，根据界面上指定的查询条件返回一个 Filter 枚举器；GetMerit，从 Filter 的属性集中获取 Merit 值。实现参考如下：

```
IEnumMoniker *CMapperDlg::GetFilterEnumerator(void)
{
    // 首先刷新几个布尔变量
    BOOL bExactMatch = m_bExactMatch.GetCheck();  // 是否严格匹配
    BOOL bAtLeastOneInputPin = m_bAtLeastOneInputPin.GetCheck();
                                            // 至少一个输入 Pin
    BOOL bAtLeastOneOutputPin = m_bAtLeastOneOutputPin.GetCheck();
                                            // 至少一个输出 Pin
    BOOL bIsRenderer = m_bIsRenderer.GetCheck();
                                            // 是不是 Renderer Filter
    // 得到 Filter 的最小 Merit 值
    DWORD dwMerit = (DWORD)
    m_ComboMerit.GetItemData(m_ComboMerit.GetCurSel());
    // 得到指定的 Pin 的类型
    const GUID *pInCat  = (const GUID *)
        m_ComboInputCat.GetItemDataPtr(m_ComboInputCat.GetCurSel());
    const GUID *pOutCat = (const GUID *)
        m_ComboOutputCat.GetItemDataPtr(m_ComboOutputCat.GetCurSel());
    // 媒体类型
    DWORD dwInputTypes=0, dwOutputTypes=0;
    GUID arrayInput[4], arrayOutput[4];
    // 初始化
    for (int i=0; i<4; i++)
    {
        arrayInput[i]  = GUID_NULL;
        arrayOutput[i] = GUID_NULL;
    }
    // 得到指定的媒体类型
    FillTypeArray(m_ListInputMajor, m_ListInputMinor, arrayInput, 0,
                  &dwInputTypes);
    if (dwInputTypes)
        FillTypeArray(m_ListInputMajor2, m_ListInputMinor2, arrayInput, 2,
                      &dwInputTypes);
    FillTypeArray(m_ListOutputMajor, m_ListOutputMinor, arrayOutput, 0,
                  &dwOutputTypes);
    if (dwOutputTypes)
        FillTypeArray(m_ListOutputMajor2, m_ListOutputMinor2,
                      arrayOutput, 2, &dwOutputTypes);
    IEnumMoniker *pEnum;
    HRESULT hr=S_OK;
    // 根据所有查询条件，得到一个枚举器
```

```
    hr = m_pMapper->EnumMatchingFilters(
            &pEnum,
            0,                      // Reserved
            bExactMatch,            // Use exact match?
            dwMerit,                // Minimum merit
            bAtLeastOneInputPin,
            dwInputTypes,// Number of major type/subtype pairs for input
            arrayInput, // Array of major type/subtype pairs for input
            NULL,                   // Input medium
            pInCat,                 // Input pin category
            bIsRenderer,            // Must be a renderer?
            bAtLeastOneOutputPin,
            dwOutputTypes, // Number of major type/subtype pairs for output
            arrayOutput, // Array of major type/subtype pairs for output
            NULL,                   // Output medium
            pOutCat);               // Output pin category
    if (FAILED(hr))
        return 0;
    else
        return pEnum;
}
// 从 Filter 的属性集中获得 Filter 的 Merit 值
HRESULT CMapperDlg::GetMerit(IPropertyBag *pPropBag, DWORD *pdwMerit)
{
    HRESULT hr;
    IAMFilterData *pData=0;
    *pdwMerit = 0;
    VARIANT varFilData={0};
    varFilData.vt = VT_UI1 | VT_ARRAY;
    varFilData.parray = 0;     // docs say to zero this
    BYTE *pbFilterData = 0;    // 0 if not read
    DWORD dwcbFilterDAta = 0;  // 0 if not read
    // 得到描述 Filter 的一个数据块
    hr = pPropBag->Read(L"FilterData", &varFilData, 0);
    if(SUCCEEDED(hr))
    {
        ASSERT(varFilData.vt == (VT_UI1 | VT_ARRAY));
        dwcbFilterDAta = varFilData.parray->rgsabound[0].cElements;
        // Access the filter data
        HRESULT hrTmp = SafeArrayAccessData(varFilData.parray, (void **)
                        &pbFilterData);
        ASSERT(hrTmp == S_OK);
        ASSERT(pbFilterData);
    }
    else
    {
        ASSERT(dwcbFilterDAta == 0 && pbFilterData == 0);
        return E_FAIL;
    }
    // Get the IAMFilterData interface for parsing the filter data
```

```
        hr = m_pMapper->QueryInterface(__uuidof(IAMFilterData), (void **)
             &pData);
        if(SUCCEEDED(hr))
        {
            BYTE *pb=0;
            // 对描述的 Filter 数据块进行分析，获取其中 Merit 部分
            hr = pData->ParseFilterData(pbFilterData, dwcbFilterDAta, &pb);
            if(SUCCEEDED(hr))
            {
                REGFILTER2 *pFil = ((REGFILTER2 **)pb)[0];
                if(pbFilterData)
                {
                    HRESULT hrTmp = SafeArrayUnaccessData(varFilData.parray);
                    ASSERT(hrTmp == S_OK);
                    hrTmp = VariantClear(&varFilData);
                    ASSERT(hrTmp == S_OK);
                }
                // 从 REGFILTER2 数据结构中获取 Merit 值
                if (pFil)
                    *pdwMerit = pFil->dwMerit;
                // Free the memory allocated by ParseFilterData
                if (pb)
                    CoTaskMemFree(pb);
            }
            // Release the IAMFilterData interface
            pData->Release();
        }
        return hr;
    }
```

16.3 DMO 枚举例子

这一节，将介绍一个 DMO 枚举例子的实现，它位于 SDK 安装目录 Samples\C++\DirectShow\Misc\DMOEnum 下。

16.3.1 实现的功能

这是一个 DMO 的枚举程序，包括枚举所有 DMO 类型目录，以及指定目录下的所有 DMO 组件。同时，还可以查看到指定 DMO 组件的各种特性（比如输入和输出流的各种属性，提供的首选媒体类型等）。运行 DMOEnum 程序，如图 16.3 所示。

在图 16.3 中，DMO Categories 框中列出了所有的 DMO 类型目录，它右边的列表框显示了指定目录下的所有 DMO 组件。其他的界面元素都用来描述当前选中的 DMO 组件的各种属性，比如选中 Input 或 Output 中的复选框，则表示当前 DMO 具有这种属性，如果没选中则不具有这种属性等。

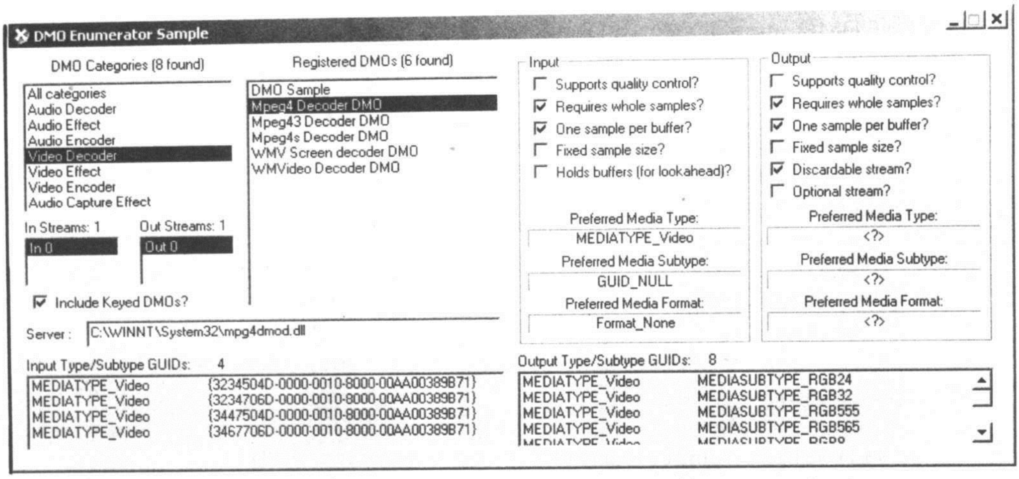

图 16.3　DMOEnum 程序的界面

16.3.2　实现要点

DMO 组件的枚举依赖于 API 函数 DMOEnum。通过这个函数，可以为指定的类型目录创建一个 DMO 枚举器，然后调用 IEnumDMO 的接口方法就可以得到注册在该目录下的所有 DMO 组件。

可以看到，DMOEnum 也是一个基于对话框的 MFC 程序。在 CDMOEnumDlg::OnSelchangeListDevices 函数中进行了对指定的类型目录的枚举工作：

```
void CDMOEnumDlg::OnSelchangeListDevices()
{
    HRESULT hr;
    IEnumDMO *pEnum = NULL;
    DWORD dwFlags = m_bCheckKeyed.GetCheck() ? DMO_ENUMF_INCLUDE_KEYED : 0;
    // 得到当前选中的 DMO 类型目录索引
    int nItem = m_DeviceList.GetCurSel();
    const GUID *clsid;
    // 得到指定目录的 CLSID
    clsid = dmo_categories[nItem].pclsid;
    ClearTypeLists();
    ReleaseDMO();
    // 使用 API 函数 DMOEnum，为指定的目录创建一个枚举器
    hr = DMOEnum(*clsid, dwFlags, 0, NULL, 0, NULL, &pEnum);
    if (FAILED(hr))
        return;
    // 枚举指定目录下的所有 DMO 组件
    hr = EnumDMOs(pEnum);
    // 自动选中枚举得到的第一个 DMO 组件
    m_FilterList.SetCurSel(0);
    // 刷新当前选中的 DMO 组件的各种属性
    OnSelchangeListFilters();
    SAFE_RELEASE(pEnum);
}
```

下面再来看 CDMOEnumDlg::EnumDMOs 函数的实现过程：

```
HRESULT CDMOEnumDlg::EnumDMOs(IEnumDMO *pEnumCat)
{
    HRESULT hr=S_OK;
    ULONG cFetched=0;
    int nFilters=0;
    WCHAR *wszName=0;
    CLSID clsid;
    // 清除 DMO 列表框的内容
    ClearFilterList();
    // If there are no filters of a requested type, show default string
    if (!pEnumCat)
    {
        m_FilterList.AddString(TEXT("<< No entries >>"));
        SetNumFilters(nFilters);
        return S_FALSE;
    }
    // 枚举所有 DMO 组件，并加入到 DMO 列表框中
    while(pEnumCat->Next(1, &clsid, &wszName, &cFetched) == S_OK)
    {
        nFilters++;
        // 获取 DMO 组件的名字（如果没有名字，则给它临时指定一个）
        CString str(wszName);
        if (str.GetLength() == 0)
            str = TEXT("<< Invalid DMO name >>\0");
        // 将枚举得到的 DMO 组件名字加入到列表框中
        AddFilter(str, &clsid);
        CoTaskMemFree(wszName);
        wszName = 0;
    }
    // If no DMOs matched the query, show a default item
    if (nFilters == 0)
        m_FilterList.AddString(TEXT("<< No entries >>"));
    // Update count of enumerated filters
    SetNumFilters(nFilters);
    return hr;
}
```

另外，关于 DMO 组件的各种描述属性获取，请参考 ShowFilenameByCLSID、AddTypeInfo、ShowSelectedDMOInfo 等函数的实现。

第 17 章　媒体播放器例子

17.1　实现的功能

这一章，将介绍一个媒体播放器的例子，它位于 SDK 安装目录 Samples\C++\DirectShow\Players\PlayWndASF 下。

这个播放器理论上可以播放任何类型的媒体文件（只要系统中注册了相应的解码 Filter）。运行这个程序，然后选择一个媒体文件进行播放，如图 17.1 所示。

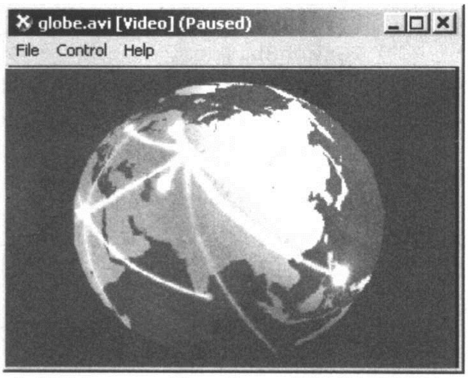

图 17.1　PlayWndASF 程序的界面

如图 17.1 所示，可以在播放的过程中，通过 Control 下的各菜单项进行各种控制，包括播放、暂停、停止，实现静音播放、单帧步进播放，调整输出视频图像大小（包括全屏播放）等。

提示

因为 PlayWndASF 例子涉及到 Windows Media 文件的播放，所以编译这个例子，需要首先安装 Windows Media Format SDK，并且配置好 VC 的 Include 和 Library 系统路径。

17.2　实 现 要 点

PlayWndASF 是一个标准的 Win32 应用程序。可以看到这个程序的入口函数 WinMain，总体的执行流程如下：CoInitializeEx | 创建主窗口 | 发送消息以弹出一个对话框选择要播放的媒体文件 | 主线程的消息循环 | CoUninitialize。

在分析这个程序的实现之前，先来看几个宏：

```
// 安全释放某个接口
#define SAFE_RELEASE(x) { if (x) x->Release(); x = NULL; }
```

```
// 判断 HRESULT 类型的返回值，如果错误则弹出消息框提示，并调用 return 返回
#define JIF(x) if (FAILED(hr=(x))) \
    {Msg(TEXT("FAILED(hr=0x%x) in ") TEXT(#x) TEXT("\n\0"), hr); return hr;}
// 判断 HRESULT 类型的返回值，如果错误则弹出消息框提示（不调用 return 返回）
#define LIF(x) if (FAILED(hr=(x))) \
    {Msg(TEXT("FAILED(hr=0x%x) in ") TEXT(#x) TEXT("\n\0"), hr);}
```

下面，主要来看播放指定媒体文件的 Filter Graph 的创建过程。先来看 PlayMovieInWindow 函数的实现。这个函数带两个参数，一个是指定的媒体文件名，另一个是控制标记（因为 PlayMovieInWindow 可能会在播放受 DRM 保护的 Windows Media 文件时被重复调用）。一般，要播放一个媒体文件时，首先调用 PlayMovieInWindow(g_szFileName, FALSE)；于是，PlayMovieInWindow 主要完成的工作有：

```
// 创建一个 Filter Graph 组件
JIF(CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
                     IID_IGraphBuilder, (void **)&pGB));
// 得到 IMediaEventEx 接口，以便于让应用程序及时处理 Filter Graph Manager 发出的
// 事件
JIF(pGB->QueryInterface(IID_IMediaEventEx, (void **)&pME));
if(SUCCEEDED(hr))
{
    // 设置一个自定义消息 WM_GRAPHNOTIFY，用于 Filter Graph Manager 对应用程序
    // 的事件通知。注意：在实际的 Filter Graph 构建之前完成这个设置，以处理 DRM 认证
    // 问题
    JIF(pME->SetNotifyWindow((OAHWND)ghApp, WM_GRAPHNOTIFY, 0));
    // 判断是不是 Windows Media 格式的文件
    if (IsWindowsMediaFile(szFile))
    {
        // 显式地创建 ASF Reader Filter
        hr = CreateFilter(CLSID_WMAsfReader, &g_pReader);
        if(FAILED(hr))
        {
            Msg(TEXT("Failed to create WMAsfWriter filter!  hr=0x%x\0"), hr);
             return hr;
        }
        // 将 ASF Reader Filter 加入到 Filter Graph 中
        hr = pGB->AddFilter(g_pReader, L"ASF Reader");
        if(FAILED(hr))
        {
          Msg(TEXT("Failed to add ASF reader filter to graph!
                  hr=0x%x\0"), hr);
          return hr;
        }
        // 创建一个密钥提供器来解锁 Windows Media SDK
        JIF(AddKeyProvider(pGB));
        // 创建一个 DRM 认证的同步事件对象
        g_hLicenseEvent = CreateEvent( NULL, FALSE, FALSE, NULL );
        if( !g_hLicenseEvent )
        {
            return E_OUTOFMEMORY;
```

```
        }
        // 装载媒体文件
        JIF(g_pReader->QueryInterface(IID_IFileSourceFilter, (void **)
            &g_pFileSource));
        hr = g_pFileSource->Load(wFile, NULL);
        // 处理 Digital Rights Management (DRM) 各种错误情况
        if(NS_E_LICENSE_REQUIRED == hr)
        {
            Msg(TEXT("This media file is protected by DRM and needs a
               license.\r\n\r\n")TEXT("Attempting to acquire a
               license...\0"));
            g_bWaitingForLicense = TRUE;  // 等待认证
            return hr;
        }
        else if(NS_E_PROTECTED_CONTENT == hr)
        {
            Msg(TEXT("This media file is protected by DRM and needs a
                   license.\r\n\r\n")
                TEXT("In order to play DRM-encoded content, you must
                   acquire a DRM stub library\r\n")
                TEXT("from Microsoft and link it with this application.
                   The default version of\r\n")
                TEXT("the WMStub.lib library does not support Digital
                   Rights Management (DRM)."));
            return hr;  // 认证失败
        }
        else if (FAILED(hr))
        {
            Msg(TEXT("Failed to load file in source filter
               (g_pFileSource->Load())!  hr=0x%x\0"), hr);
            return hr;  // 装载源文件失败
        }
        // 该 Windows Media 文件没有受 DRM 保护...将 ASF Reader Filter 的各个输
        // 出 Pin 连接下去，构建完整的播放 Filter Graph
        JIF(RenderOutputPins(pGB, g_pReader));
        // 因为 ASF Reader Filter 已经加入到了 Filter Graph，Filter Graph
        // Manager 会增加引用计数。这里可以释放应用程序上维持的一个引用计数
        g_pReader->Release();
        g_pReader = NULL;
    }
    else
    {
    // 如果指定播放的不是 Windows Media 文件，简单地使用智能连接构建 Filter Graph
        JIF(pGB->RenderFile(wFile, NULL));
    }
}
```

然后就是获取 Filter Graph 的各个控制接口，实现如下：

```
if( SUCCEEDED( hr ) )
{
```

```
    // 得到播放控制接口和随机定位接口
    JIF(pGB->QueryInterface(IID_IMediaControl, (void **)&pMC));
    JIF(pGB->QueryInterface(IID_IMediaSeeking, (void **)&pMS));
    // 得到视频相关的接口
    JIF(pGB->QueryInterface(IID_IVideoWindow, (void **)&pVW));
    JIF(pGB->QueryInterface(IID_IBasicVideo,  (void **)&pBV));
    // 得到音频相关的接口
    JIF(pGB->QueryInterface(IID_IBasicAudio, (void **)&pBA));
    // 检查媒体文件是否包含有视频内容
    CheckVisibility();
    if (!g_bAudioOnly)
    {
        // 设置视频窗口
        JIF(pVW->put_Owner((OAHWND)ghApp));
        JIF(pVW->put_WindowStyle(WS_CHILD | WS_CLIPSIBLINGS |
        WS_CLIPCHILDREN));
        JIF(InitVideoWindow(1, 1));
        GetFrameStepInterface();
    }
    else
    {
        // 这是一个纯音频文件播放器...
        JIF(InitPlayerWindow());
        EnablePlaybackMenu(TRUE, AUDIO);
    }
    // 完成其他的窗口初始化
    CheckSizeMenu(ID_FILE_SIZE_NORMAL);
    ShowWindow(ghApp, SW_SHOWNORMAL);
    UpdateWindow(ghApp);
    SetForegroundWindow(ghApp);
    g_bFullscreen = FALSE;
    UpdateMainTitle();
#ifdef REGISTER_FILTERGRAPH
    // 将当前 Filter Graph 注册到 ROT，以使 GraphEdit 能够查看到实际的连接图
    hr = AddGraphToRot(pGB, &g_dwGraphRegister);
    if (FAILED(hr))
    {
        Msg(TEXT("Failed to register filter graph with ROT! hr=0x%x"), hr);
        g_dwGraphRegister = 0;
    }
#endif
    // 播放 Filter Graph
    JIF(pMC->Run());
    // 更新当前的状态
    g_psCurrent=Running;
    SetFocus(ghApp);
}
```

如何判断一个文件是不是 Windows Media 格式的文件？

已知 Windows Media 格式的文件一般有三种扩展名：.asf、.wma 和.wmv。可以通过文件的扩展名来简单地判断一个文件是不是 Windows Media 文件。实现如下：

```
BOOL IsWindowsMediaFile(LPTSTR lpszFile)
{
    TCHAR szFilename[MAX_PATH];
    // Copy the file name to a local string and convert to lowercase
    _tcsncpy(szFilename, lpszFile, NUMELMS(szFilename));
    szFilename[MAX_PATH-1] = 0;
    _tcslwr(szFilename);
    if (_tcsstr(szFilename, TEXT(".asf")) ||
        _tcsstr(szFilename, TEXT(".wma")) ||
        _tcsstr(szFilename, TEXT(".wmv")))
        return TRUE;
    else
        return FALSE;
}
```

下面再来看应用程序对 Filter Graph Manager 上的事件处理，即 WM_GRAPHNOTIFY 的消息响应（参见 HandleGraphEvent 函数）：

```
HRESULT HandleGraphEvent(void)
{
    LONG evCode, evParam1, evParam2;
    HRESULT hr=S_OK;
    // Make sure that we don't access the media event interface
    // after it has already been released.
    if (!pME)
        return S_OK;
    // 一次 while 循环处理完事件队列中的所有事件
    while(SUCCEEDED(pME->GetEvent(&evCode, (LONG_PTR *) &evParam1,
                    (LONG_PTR *) &evParam2, 0)))
    {
        // 当媒体文件播放完毕，重新开始播放
        if(EC_COMPLETE == evCode)
        {
            LONGLONG pos=0;
            // 定位到文件头
            hr = pMS->SetPositions(&pos, AM_SEEKING_AbsolutePositioning ,
                                   NULL, AM_SEEKING_NoPositioning);
            if (FAILED(hr))
            {
                // 有些少数的 Filter 不支持 IMediaSeeking 接口
                // 于是可以通过调用 Stop 后再调用 Run 来实现重新播放
                if (FAILED(hr = pMC->Stop()))
                {
                  Msg(TEXT("Failed(0x%08lx) to stop media clip!\r\n"),hr);
                  break;
                }
                if (FAILED(hr = pMC->Run()))
                {
```

```
                    Msg(TEXT("Failed(0x%08lx) to reset media clip!\r\n"),hr);
                     break;
                }
            }
        }
        else if(EC_ERRORABORT == evCode) // Filter Graph 运行过程中发生了错误
        {
            Msg(TEXT("Playback error (EC_ERRORABORT - 0x%08lx). Aborting
            ...\r\n"), evParam1);
            if (FAILED(hr = pMC->Stop()))
                Msg(TEXT("Failed(0x%08lx) to stop media clip!\r\n"), hr);
            hr = (HRESULT) evParam1;
            break;
        }
        else if( EC_WMT_EVENT == evCode ) // 处理 Windows Media 相关的事件
        {
            AM_WMT_EVENT_DATA *pEventData = (AM_WMT_EVENT_DATA *)evParam2;
            hr = HandleWMTEvent(pEventData, evParam1);
        }
    }
    // Free memory associated with callback
    if( FAILED(pME->FreeEventParams(evCode, evParam1, evParam2)) )
    {
        CloseClip();
    }
    return hr;
}
```

可以看到，对于 Windows Media 文件的 DRM 认证处理在 HandleWMTEvent 函数中实现：

```
HRESULT HandleWMTEvent(AM_WMT_EVENT_DATA *pEventData, LONG evParam1)
{
    HRESULT hr=S_OK;
    // 如果 Windows Media 文件使用了 DRM 保护，但播放器没有使用 DRM 存根库，
    // 则 pEventData 参数将是一个 NULL 值
    // 必须使用一个适当的 DRM 存根库重新编译生成一个播放器程序
    if( NULL == pEventData )
    {
        Msg(TEXT("Received EC_WMT_EVENT but evParam2 is NULL!
            Aborting...\r\n") );
        return E_FAIL;
    }
    switch( evParam1 )
    {
        case WMT_NO_RIGHTS:  // 无权播放受 DRM 版本 1 保护的 Windows Media 文件
            Msg(TEXT("Received WMT_NO_RIGHTS! (hrStatus = 0x%x)\0"),
                    pEventData->hrStatus);
            if( pEventData->pData )
            {
                hr = HandleNoRights(pEventData->hrStatus, (WCHAR
```

```
                            *)pEventData->pData);
            }
            else
            {
                hr = E_FAIL;
            }
            break;
        case WMT_NO_RIGHTS_EX://无权播放受 DRM 版本 7 保护的 Windows Media 文件
            // license acquisition?
            Msg(TEXT("Received WMT_NO_RIGHTS_EX! (hrStatus = 0x%x)\r\n"),
                    pEventData->hrStatus);
            if( pEventData->pData )
            {
                hr = HandleNoRightsEx(pEventData->hrStatus, (WM_GET_LIC-
                                    ENSE_DATA *)pEventData->pData);
            }
            else
            {
                hr = E_FAIL;
            }
            break;
        case WMT_ACQUIRE_LICENSE:  // DRM 认证完成（可能成功，也可能失败）
            Msg(TEXT("Received WMT_ACQUIRE_LICENSE! (hrStatus =
                    0x%x)\r\n"), pEventData->hrStatus);
            if( pEventData->pData )
            {
                hr = HandleAcquireLicense(pEventData->hrStatus,
                        (WM_GET_LICENSE_DATA *)pEventData->pData);
            }
            else
            {
                hr = E_FAIL;
            }
            break;
    }
    return hr;
}
```

具体的 DRM 认证过程，还可以参考 HandleNoRights、HandleNoRightsEx 和 HandleAcquireLicense 函数的实现。另外，在 HandleAcquireLicense 函数中，当成功进行了 DRM 认证后，会调用 PlayMovieInWindow(g_szFileName, TRUE) 重新装载媒体文件。此时 PlayMovieInWindow 完成的工作将是：

```
hr = g_pFileSource->Load(wFile, NULL);  // Source Filter 重新装载媒体文件
if( SUCCEEDED( hr ) )
{
    Msg(TEXT("Successfully loaded file after DRM license acquisition!"));
    // Render the output pins of the ASF reader to build the
    // remainder of the graph automatically
    JIF(RenderOutputPins(pGB, g_pReader));
```

```
        // Since the graph is built and the filters are added to the graph,
        // the WM ASF reader interface can be released.
        g_pReader->Release(); // not really necessary
        g_pReader = NULL;
    }
    else
    {
        Msg(TEXT("Failed to Load file after acquiring license!  hr=0x%x\0"), hr);
    }
    // 然后就是执行获取 Filter Graph 的各个控制接口的程序块...
    // ...
```

接下来，看一下密钥提供器的实现过程。可以看到 keyprovider.h 和 keyprovider.cpp 两个文件定义和实现了一个 CKeyProvider 类。并且，播放器定义了一个 CKeyProvider 类的全局对象实例：

```
CKeyProvider prov;
```

再看 AddKeyProvider 函数的实现，如下：

```
HRESULT AddKeyProvider(IGraphBuilder *pGraph)
{
    HRESULT hr;
    // 增加一个引用计数，防止 COM 调用 delete 删除 prov 对象
    // 因为 prov 不是通过 new 创建的
    prov.AddRef();

    // 从 Filter Graph Manager 上获得 IObjectWithSite 接口
    IObjectWithSite* pObjectWithSite = NULL;
    hr = pGraph->QueryInterface(IID_IObjectWithSite,
(void**)&pObjectWithSite);
    if (SUCCEEDED(hr))
    {
        // 设置密钥提供器对象指针给 Filter Graph Manager
        // Filter Graph Mananger 会调用 IServiceProvider::QueryService 进行
        // SDK 的解锁。只有解锁成功后，才能继续构建 Filter Graph
        hr = pObjectWithSite->SetSite((IUnknown *) (IServiceProvider *)
                                          &prov);
        pObjectWithSite->Release();
    }
    return hr;
}
```

因为支持受 DRM 保护的 Windows Media 文件，播放器创建 Filter Graph 的过程变得复杂了很多。下面，就来关注一下其他方面的问题。

使用智能连接，完成给定 Filter 的所有输出 Pin 的连接

参见 RenderOutputPins 函数的实现：

```
HRESULT RenderOutputPins(IGraphBuilder *pGB, IBaseFilter *pFilter)
{
```

```
    HRESULT          hr = S_OK;
    IEnumPins *      pEnumPin = NULL;
    IPin *           pConnectedPin = NULL, * pPin = NULL;
    PIN_DIRECTION    PinDirection;
    ULONG            ulFetched;
    // 枚举 Filter 上的所有 Pin
    hr = pFilter->EnumPins(&pEnumPin);
    if(SUCCEEDED(hr))
    {
        while (S_OK == (hr = pEnumPin->Next(1L, &pPin, &ulFetched)))
        {
            // 判断当前 Pin 是否已经完成连接，因为不须再处理这种 Pin
            hr = pPin->ConnectedTo(&pConnectedPin);
            if (pConnectedPin)
            {
                pConnectedPin->Release();
                pConnectedPin = NULL;
            }

            // 如果当前 Pin 没有连接，并且是输出 Pin，则调用 IGraphBuilder::
            // Render 进行智能连接
            if (VFW_E_NOT_CONNECTED == hr)
            {
                hr = pPin->QueryDirection(&PinDirection);
                if ((S_OK == hr) && (PinDirection == PINDIR_OUTPUT))
                {
                    hr = pGB->Render(pPin);
                }
            }
            pPin->Release();
            // If there was an error, stop enumerating
            if (FAILED(hr))
                break;
        }
    }
    // Release pin enumerator
    pEnumPin->Release();
    return hr;
}
```

检查给定的媒体文件是否含有音频，或是否含有视频

参见 CheckVisibility 函数的实现：

```
// 当完成创建播放该媒体文件的 Filter Graph 后，调用 CheckVisibility 判断
// 判断结果保存到布尔变量 g_bAudioOnly 中
void CheckVisibility(void)
{
    long lVisible;
    HRESULT hr;
    if ((!pVW) || (!pBV))
```

```
    {
     // 播放纯音频文件的 Filter Graph 不会实现 IVideoWindow 和 IBasicVideo 接口
        g_bAudioOnly = TRUE;
        return;
    }
    else
    {
        // Clear the global flag
        g_bAudioOnly = FALSE;
    }
// 另外一种情况，媒体文件实际包含有视频内容，只是系统中没有相应的解码 Filter
// 此时 pVW->get_Visible 调用将失败
    hr = pVW->get_Visible(&lVisible);
    if (FAILED(hr))
    {
        if (hr == E_NOINTERFACE)
        {
            g_bAudioOnly = TRUE;
        }
        else
        {
            Msg(TEXT("Failed(%08lx) in pVW->get_Visible()!\r\n"), hr);
        }
    }
}
```

实现媒体文件的静音播放

参见 ToggleMute 函数实现，主要是调用接口函数 IBasicAudio::put_Volume。注意最高音量 VOLUME_FULL 定义为 0L，静音 VOLUME_SILENCE 定义为-10000L。

实现全屏播放

参见 ToggleFullScreen 函数实现，主要是调用接口函数 IVideoWindow::put_FullScreen-Mode。注意在全屏模式下，须将视频窗口接收到的键盘、鼠标消息发送给主窗口，即在进入全屏模式之前应该先调用 pVW->put_MessageDrain((OAHWND) ghApp)；否则全屏模式下将无法正确响应键盘、鼠标消息。

实现步进播放（单帧或多帧）

参见 StepFrames 函数的实现：

```
// 之前首先要从 Filter Graph Manager 上获得 IVideoFrameStep 接口，并且调用
// IVideoFrameStep::CanStep 进行步进功能的确认（参见 GetFrameStepInterface 函数）
HRESULT StepFrames(int nFramesToStep)
{
    HRESULT hr=S_OK;
    // 判断 IVideoFrameStep 接口对象指针是否有效
    if (pFS)
    {
        // 首先调用 IVideoFrameStep::CanStep 进行确认
```

```
        if ((hr = pFS->CanStep(nFramesToStep, NULL)) == S_OK)
        {
            // Filter Graph 必须先处于暂停状态
            if (g_psCurrent != State_Paused)
                PauseClip();
            // 实际执行步进功能，跳跃前进 nFramesToStep 帧
            hr = pFS->Step(nFramesToStep, NULL);
        }
    }
    return hr;
}
```

第 4 部分　开放源码分析

第 18 章　MPEG 流的网络客户端播放

18.1　需 求 定 义

大家知道，在网络上传输音频、视频等多媒体数据，目前主要有下载和流式传输两种方案。下载的主要缺点是，必须等全部内容传输完毕，然后才能在本地机器打开；而采用流式传输方案，声音、影像或动画等时基媒体由音视频服务器向用户计算机连续、实时地传送，用户不必等到整个文件全部下载完毕，只需经过几秒或十几秒的启动延时即可进行观看。

流式传输广泛应用于网络视频点播，以及视频监控等领域。下面，将以网络传送 MPEG 流、客户端实时解码播放为例，介绍一下这个流式传输的解决方案。

18.2　解 决 方 案

多媒体的流式传输实际上牵涉到两方面的技术。其一就是服务器与客户端的通信技术，包括多媒体数据的传输、命令控制等；其二就是客户端对接收到的多媒体流实时解码后播放的技术。显然，网络通信可以使用 Windows Socket 技术，多媒体流的解码播放可以使用 DirectShow 技术。

18.2.1　Windows Socket 网络传输技术

已知 MPEG 流是连续的，它的同步点不能随意选择。因此，为了传送这种数据，必须使用面向连接的可靠的传输协议（TCP）。这个通信过程如图 18.1 所示。

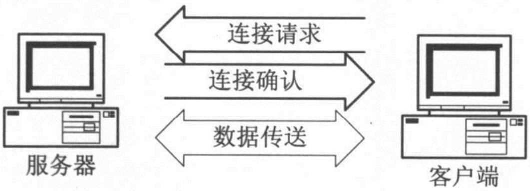

图 18.1　计算机之间的 Socket 通信过程

如图 18.1，首先是客户端使用一个 Socket 向服务器发送一个连接请求。服务器一般有一个 Socket 专门用于监听网络连接，当收到其他计算机的连接请求时，则根据实际情况接受或者拒绝连接；如果服务器接受连接，则会产生一个新的 Socket，与客户端的 Socket 建立一对连接，而实际的数据传送就是在这对 Socket 连接上完成的。

MFC 提供了两个类：CAsyncSocket 和 CSocket，用以方便开发者使用 Socket 特性。这两个类的继承关系如图 18.2 所示。

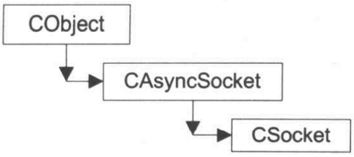

图 18.2　CAsyncSocket 和 CSocket 的类继承关系

CAsyncSocket 从较低层次封装了 Windows Socket API，并且通过内建一个（隐藏的）窗口，实现了适合 Windows 应用的异步机制。CSocket 类从 CAsyncSocket 中派生而来，更简化了开发者对 Socket 的应用。但是，这两个类并不是线程安全的；如果要在多线程环境下应用 Socket 特性，建议还是自己封装 Socket API 函数。

另外，实际的网络应用环境可能是很复杂的。在数据流传送过程中，可能还要实现其他更多的控制，比如调整服务器数据发送速率、响应客户端各种命令等。这些控制也要使用 Socket 通信，不过可以使用数据报（UDP）的形式来实现。

18.2.2　DirectShow 技术应用

客户端的多媒体数据解码，使用 DirectShow 技术是最方便的了。也就是说，需要写一个能够接收服务器发来的数据、并提供给 Filter Graph 中其他的 Filter 解码的 Source Filter。这种 Filter Graph 可以如图 18.3 所示构建。

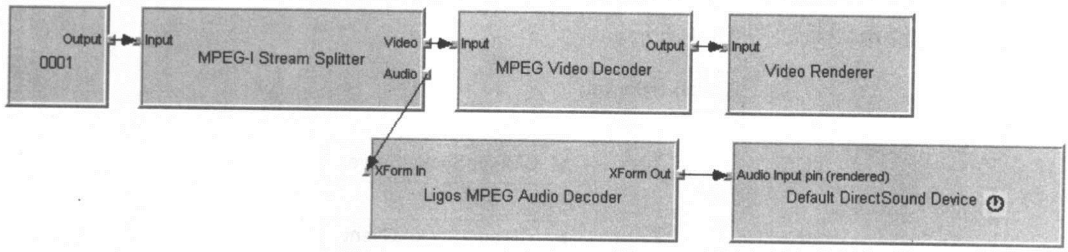

图 18.3　接收并播放 MPEG1 流的 Filter Graph

如图 18.3，标识为“0001”的 Source Filter 用于接收网络上发来的 MPEG1 数据，且工作在拉模式下。（这个 Source Filter 的改写可以参考 SDK 中的一个例子，它的路径为 \samples\C++\DirectShow\Filters\Async\MEMFile。）

可以知道，DirectShow 提供了 MPEG1 解码的所有 Filter，包括 MPEG-1 Stream Splitter、MPEG Video Decoder 和 MPEG Audio Decoder；但是 MPEG-1 Stream Splitter 只能工作在拉模式下，这样不得不把 Source Filter 也设计成拉模式的。（不过，如果有第三方的可以工作在推模式下的 Splitter Filter，则完全可以把 Source Filter 也设计成推模式的，即接到网络上发来的数据后主动传送给 Splitter Filter，而不是被动地等待 Splitter 去读取；推模式的 Source Filter 实现起来要方便一点。）

18.2.3　一种双缓冲队列技术

客户端的数据接收也需要一定的技巧。首先，必须让 Source Filter 接收一定数量的数据，并缓存起来。因为在 Source Filter 与 Splitter 连接的过程中，会从 Source Filter 中读取

一部分数据，以获得数据的格式描述。（否则，Filter Graph 是无法完成连接的。）而当完整的 Filter Graph 构建完成，并且处于运行状态后，Source Filter 必须动态地接收数据，并持续地把新的数据提供给 Splitter。

这里，可以使用一种叫双缓冲队列的技术。大致的工作原理是：建立两个队列，一个是空闲的缓冲队列 PoolList，用以接收存放数据；另一个是尚未处理的数据缓冲队列 DataList，等待 Splitter 读取。当 Source Filter 使用 Socket 接收到数据后，从 PoolList 队列的头上取出一个缓冲块，存放数据，然后将这个缓冲块加入到 DataList 的尾部，等待 Splitter 的读取。而在 Splitter 要求读取时，Source Filter 会从 DataList 队列的头上取出一个缓冲块，读取数据，再将读完的缓冲块加到 PoolList 的尾部，“回收再利用”，等待再一次的数据接收。

18.3 源码分析

下载地址中提供了所有实现的源代码。代码实现分两部分：服务器程序（MediaServer）和客户端程序（MediaClient）；可以使用 VC 打开 Daisy.dsw 文件，同时浏览这两个项目的实现。

下面先来介绍一下这两个项目的一些公共的实现。比如 Socket API 的封装，设计了一系列的类：CListenSocket、CWorkerSocket、CMediaSocketServer、CMediaSocketClient 等。这些类的继承关系如图 18.4 所示。

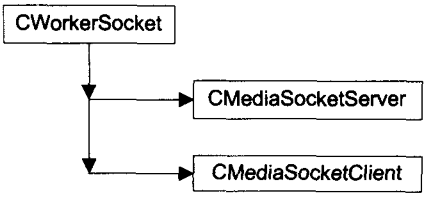

图 18.4 Socket 封装类的继承关系

CListenSocket 类主要用于在服务器端监听客户端计算机的网络连接（参见 CListenSocket.cpp 的实现）；CWorkerSocket 类封装了各种 Socket 操作，主要用于实际的数据传送和接收，并且使用 TCP 协议（参见 CWorkerSocket.cpp 的实现）；CMediaSocketServer 类从 CWorkerSocket 中派生，主要用于服务器端的数据发送（参见 CMediaSocketServer.cpp 的实现）；CMediaSocketClient 类也从 CWorkerSocket 中派生，主要用于客户端的数据接收（参见 CMediaSocketClient.cpp 的实现）。

> **提示** 还有一个跟 Socket 相关的类 CLocalMachine，主要用于获取本地计算机的名字，以及 IP 地址（参见 CLocalMachine.cpp 的实现）。

服务器与客户端之间还有一个 Socket 通信协议问题。为了简单起见，将控制通信与数据通信合并在一起。设计思路是，在服务器端首先将连续的 MPEG 流分成一块一块小包的负载数据，并且加上一个信息头一起发送；在客户端根据信息头描述，将小包的负载

数据进行拼装。信息头和负载数据定义如下：

```
// 信息头定义
typedef struct
{
    unsigned int    nMsgType:8;      // 负载类型，用 8 位表示
    unsigned int    nDataSize:16;    // 负载数据长度，用 16 位表示
} MSG_HEADER, *PMSG_HEADER;
// 负载数据
typedef struct
{
    CHAR MPEGData[2324];       // MPEG1 流一个 Pack 的长度为 2324 字节
} MPEG1_PACK, *PMPEG1_PACK;
```

Socket 数据传送结构如图 18.5 所示。

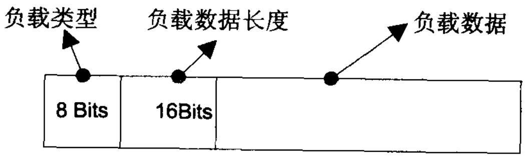

图 18.5　Socket 数据传送结构

此外，还定义了几种负载类型，如下：

```
#define DATA_REQUEST            0X00     // 客户端要求服务器发送数据
#define DATA_REFUSED            0X01     // 拒绝数据发送（比如客户端认证失败）
#define DATA_MEDIA              0X02     // 负载数据
#define DISCONNECT_REQUEST      0X03     // 客户端要求服务器断开连接
#define RECV_EXIT               0x20 // 由服务器发向客户端，表示已经停止数据发送
#define RECV_EXIT_REQUEST       0x21     // 客户端发出，要求服务器停止数据发送
```

下面，来看服务器程序的实现。这是一个基于对话框的 MFC 程序。在主对话框类 CMediaServerDlg 上，定义了如下几个成员变量：

```
protected:
    CListenSocket * m_pListener;       // 监听 Socket 对象指针
    CWorkerSocket * m_pWorkerSock;     // 数据传送 Socket 对象指针
    CString         m_strSourceFile;   // 源文件
```

并且在初始化函数 CMediaServerDlg::OnInitDialog 中的实现如下：

```
// 取得服务器的名字以及 IP 地址
char  host[255], hostIP[255];
CLocalMachine  myhost;
myhost.GetIPAddress(hostIP, host);
m_host.SetWindowText(host);
m_hostIP.SetWindowText(hostIP);

// 创建一个监听的 Socket
bool  pass = true;
m_pListener = new CListenSocket();
```

```
// 监听端口为 BASE_SOCKET_PORT，被定义为 10025
pass = m_pListener->Create(BASE_SOCKET_PORT);
if (pass)
{
    // 开始监听是否要网络计算机连接
    pass = m_pListener->StartListen();
}

// 选择一个 MPEG 源文件，以便演示数据发送
if (pass)
{
    OnButtonSelect();
}
```

其中，OnButtonSelect 函数的实现如下：

```
void CMediaServerDlg::OnButtonSelect()
{
    CFileDialog  dlg(TRUE, NULL, NULL, OFN_HIDEREADONLY |
                     OFN_OVERWRITEPROMPT, "Mpeg Files (*.mpg;*.mpeg)
                     |*.mpg; *.mpeg|", NULL);
    if (dlg.DoModal() == IDOK)
    {
            m_strSourceFile = dlg.GetPathName();
            ListenOnConnection();
    }
}
```

另外，还定义了一个消息 WM_NEW_SOCKET，用于当监听 Socket 接受一个客户端连接时通知主对话框（参见 CListenSocket::Accept 函数的实现）。主对话框上应该添加这个消息的响应，如下：

```
BEGIN_MESSAGE_MAP(CMediaServerDlg, CDialog)
    //{{AFX_MSG_MAP(CMediaServerDlg)
    // ...
    //}}AFX_MSG_MAP
    ON_MESSAGE(WM_NEW_SOCKET, ListenOnConnection)
END_MESSAGE_MAP()
void CMediaServerDlg::ListenOnConnection()
{
    if (m_pListener)
    {
            // 检查是否有客户端连接
            if (m_pListener->m_ClientList.GetCount())
            {
                    SOCKET pSock = (SOCKET) m_pListener->m_ClientList.
                                   GetHead();
                    if (m_pWorkerSock == NULL)
                            m_pWorkerSock = new CMediaSocketServer();
                    // 为数据通信做准备
                    m_pWorkerSock->Attach(pSock);
```

```
                // 设置源文件
                ((CMediaSocketServer *)m_pWorkerSock)->SetSourceFile
                (m_strSourceFile);
                // 启动接收线程，监听来自客户端的控制命令
                m_pWorkerSock->StartReceiving();
            }
        }
    }
```

此时，服务器处于就绪状态。当服务器发出命令，服务器再根据命令的类型执行相应的操作，比如数据发送、停止、断开连接等。服务器端具体的命令接收，参见函数 CMediaSocketServer::ReceivingLoop；数据的发送过程，参见函数 CMediaSocketServer::SendingLoop。

服务器程序实现起来比较简单。接下去，重点分析一下客户端程序。首先是 Source Filter 的开发。这是一个应用程序内的 Filter，并且是参考 SDK 中的 MEMFile 例子修改过来的：直接从 SDK 例子中将 asyncrdr.cpp、asyncrdr.h、asyncio.cpp、asyncio.h 四个文件拷贝过来，然后在 MemFilter.h 文件中实现一个 CAsyncStream 的派生类 CMemStream，以及一个 Filter 类 CMemReader（从 CAsyncReader 类中派生）。这里重点是 CMemStream 类的实现：

```
class CMemStream : public CAsyncStream
{
private:
    CCritSec        m_csLock;             // 数据操作的同步对象
    CDataAdmin *    m_pDataList;          // 数据接收队列管理器
    ULONG           m_ulPositionInPack;   // 当前缓冲块内的偏移量
    LONGLONG        m_llLength;           // 当前可以获得的数据总长度
    DWORD           m_dwTimeStart;
public:
    CMemStream(CDataAdmin* inBuffer)
    {
        // 成员变量初始化
        m_pDataList         = inBuffer;
        m_ulPositionInPack  = 0;
        m_llLength          = 0;
    }
    HRESULT SetPointer(LONGLONG llPos)   // 没有实际作用
    {
        return S_OK;
    }
    // 当 Splitter 要求数据时，必须在这个函数中提供
    HRESULT Read(PBYTE pbBuffer,             // 装载读出数据的缓存地址
                 DWORD dwBytesToRead,        // 要求读取的字节数
                 BOOL bAlign,                // 数据对齐方式
                 LPDWORD pdwBytesRead)       // 实际读取的字节数
    {
        if (m_pDataList == NULL)
            return S_FALSE;
```

```
CAutoLock      lck(&m_csLock);
DWORD          dwHaveRead = 0;   // 已经读取的字节数
PMPEG1_PACK    pPack   = NULL;
      // 如果已经读取的字节数没有达到要求的字节数，则继续 while 循环
while (dwHaveRead < dwBytesToRead)
{
      // 如果剩下还要读取的字节数大于当前缓冲块中剩余的有效字节数
      // 则直接将当前缓冲块的剩余数据读出
      if (dwBytesToRead - dwHaveRead >= MPEG1_PACK_SIZE -
          m_ulPositionInPack)
      {
            // 首先得到缓冲块的指针
            pPack = m_pDataList->GetDataBuffer();
            if (pPack != NULL)
            {
                  CopyMemory((PVOID)(pbBuffer + dwHaveRead),
                        (PVOID)((PBYTE)(pPack) +
m_ulPositionInPack), (SIZE_T)MPEG1_PACK_SIZE - m_ulPositionInPack);
                  // 回收数据已经被完全读出的缓冲块，等待下次接收数据
                  m_pDataList->ReleaseDataBuffer(pPack);
                  // 修改已经读取的字节数
                  dwHaveRead += MPEG1_PACK_SIZE -
                              m_ulPositionInPack;
                  // 缓存块内的偏移置 0
                  m_ulPositionInPack = 0;
            }
            else if (m_pDataList->IsFlushing())
            {
                  return E_FAIL;   // 跳出 while 循环
            }
            else
            {
                  Sleep(10);       // 等待接收更多的数据
            }
      }
      else
      {
            // 拷贝当前缓冲块内的部分数据
            pPack = m_pDataList->PointToDataHead();
            if (pPack != NULL)
            {
                  // 记住缓存块内的偏移
                  m_ulPositionInPack = dwBytesToRead -
                                 dwHaveRead;
                  CopyMemory((PVOID)(pbBuffer + dwHaveRead),
                   (PVOID)(pPack), (SIZE_T)m_ulPositionInPack);
                  // 修改已经读取的字节数
                  dwHaveRead += m_ulPositionInPack;
            }
            else if (m_pDataList->IsFlushing())
```

```
                    {
                        return E_FAIL;
                    }
                    else
                    {
                        Sleep(10);
                    }
                }
            }
            *pdwBytesRead = dwBytesToRead;
            return S_OK;
        }
        LONGLONG Size(LONGLONG *pSizeAvailable)
        {
            *pSizeAvailable = m_llLength;
            return 0x7fffffffff; // 不知道服务器总共要发送多少数据，则取一个很大的值
        }
        DWORD Alignment()
        {
            return 1;  // 按 1 字节对齐
        }
        void Lock()
        {
            m_csLock.Lock();
        }
        void Unlock()
        {
            m_csLock.Unlock();
        }
        void AddAvailableLength(LONGLONG inLength)
        {
            m_llLength += inLength;
        }
};
```

提示　数据接收队列管理器参见 CDataAdmin 类的实现。

下面再来看 Filter Graph 的构建过程（参见 CFilterGraph::BuildGraph 的实现）。可以看到，BuildGraph 函数首先创建了一个 Filter Graph 组件，然后获得一系列控制接口，接下来创建 Source Filter。值得注意的是，因为这是一个应用程序内实现的 Filter，不能通过 CoCreateInstance 函数去创建它，而要直接生成：

```
if (m_bInit)
{
    CMediaType    mt;
    mt.majortype  = MEDIATYPE_Stream;
    mt.subtype    = MEDIASUBTYPE_MPEG1System;   // 接收的是 MPEG1 类型的数据流
```

```
    m_pSourceStream   = new CMemStream(m_pDataList);
    m_pSourceReader   = new CMemReader(m_pSourceStream, &mt, &hr);
// Filter 对象
    m_pSourceReader->AddRef();    // 增加一个引用计数，防止 COM 的自动删除
    // 将我们的 Source Filter 加入到 Filter Graph 中
    hr = m_pGB->AddFilter(m_pSourceReader, NULL);
    if (FAILED(hr))
    {
            m_bInit = false;
    }

}
```

BuildGraph 函数最后并没有将 Source Filter 的输出 Pin 连出去，因为 Source Filter 在没有缓存一定数量的数据之前，与 Splitter 连接是不会成功的。解决办法是，启动一个等待线程；当 Source Filter 预先接收到足够的数据后，再构建完整的 Filter Graph。实现如下：

```
bool CFilterGraph::BuildGraph(void)
{
    m_bInit = false;
    // ...
    if (m_bInit)
    {
        // 启动一个等待线程
        ::AfxBeginThread((AFX_THREADPROC)CFilterGraph::WaitingThrd, this);
    }
    return m_bInit;
}
// 等待线程的执行函数体
UINT CFilterGraph::WaitingThrd(void * pParam)
{
    CFilterGraph * pGraph = (CFilterGraph *) pParam;
    if (pGraph != NULL && pGraph->m_pDataList != NULL)
    {
            // 询问数据接收队列管理器，是否已经接受到了足够的数据
            ::WaitForSingleObject(pGraph->m_pDataList->m_hBufEnough,
                                  INFINITE);
            if (!pGraph->IsRunning())
            {
                    // 开始构建完整的 Filter Graph
                    pGraph->StartGraph();
            }
    }
    return 1;
}
// 构建完成的 Filter Graph，并运行它
bool CFilterGraph::StartGraph(void)
{
    if (m_bInit)
    {
            m_bRun = true;
```

```
            // 使用智能连接，将 Source Filter 的输出 Pin 连出去
            HRESULT hr   = m_pGB->Render(m_pSourceReader->GetPin(0));
            if (FAILED(hr))
            {
                  m_bRun = false;
                  return false;
            }
            // 设置视频窗口的一些属性
            hr = m_pVW->put_Visible(OAFALSE);
            hr = m_pVW->put_Owner((OAHWND)m_hOwner);
            hr = m_pVW->put_WindowStyle(WS_CHILD | WS_CLIPSIBLINGS |
                WS_CLIPCHILDREN);
            hr = m_pVW->SetWindowPosition(m_nLeft, m_nTop, m_nWidth,
                m_nHeight);
            hr = m_pVW->put_MessageDrain((OAHWND)m_hOwner);
            hr = m_pVW->put_Visible(OATRUE);

            // 运行 Filter Graph
            hr = m_pMC->Run();
            return true;
        }

        return false;
    }
```

另外，也可以从 CMediaSocketClient::ReceivingLoop 函数的实现中，查看客户端的整个数据接收过程。可以看到，ReceivingLoop 函数使用了一个 while 循环不断地接收数据，并把数据交给队列管理器；同时对接收到的数据包进行计数（并在界面上显示），当接收到的包数达到要求预接收的数量 PACK_INIT_COUNT（定义值为 100），则设置队列管理器中的一个事件对象 m_hBufEnough 有效。上面提到的等待线程，正是检查到 m_hBufEnough 对象状态有效时，才开始真正的 Filter Graph 构建工作。

最后，来看一下客户端在应用程序上的实现。来看 CMediaClientDlg::OnInitDialog 函数：

```
    // COM 库初始化
    if (FAILED(CoInitialize(NULL)))
    {
        AfxMessageBox("COM 初始化失败！");
        return FALSE;
    }
    // 得到本机的主机名和 IP 地址
    char  host[255], hostIP[255];
    CLocalMachine  myhost;
    myhost.GetIPAddress(hostIP, host);
    m_host.SetWindowText(host);
    m_hostIP.SetWindowText(hostIP);
    m_btnConnect.SetWindowText("连 接");
    m_btnTrans.EnableWindow(FALSE);
    m_ipCtrl.SetAddress(127, 0, 0, 1);
```

```
// 创建一个数据接收队列管理器
m_pDataList = new CDataAdmin();
// 创建一个 Filter Graph 封装类对象
m_pGraph = new CFilterGraph(m_pDataList);
// 创建 Filter Graph
m_pGraph->BuildGraph();
// 设置视频窗口属性
m_pGraph->SetVideoWndOwner(m_videoWnd.GetSafeHwnd());
RECT rect;
m_videoWnd.GetClientRect(&rect);
    m_pGraph->SetWindowPosition(0, 0, rect.right - rect.left,
                               rect.bottom - rect.top);
```

然后当用户单击“连接”按钮时，主要在 CMediaClientDlg::OpenSession 函数中实现：

```
void CMediaClientDlg::OpenSession(void)
{
   SAFE_DELETE(m_pReceiver)
   // 在客户端创建一个 Socket
   m_pReceiver = new CMediaSocketClient();
   // 将数据队列管理器设置给 Socket 封装对象，便于数据接收
   m_pReceiver->SetRecvBuffer(m_pDataList);
   m_pReceiver->SetCountWnd(m_Counter.GetSafeHwnd());
   // 向服务器的 BASE_SOCKET_PORT 端口试连接（服务器 IP 地址由用户指定）
   // 已知服务器在 BASE_SOCKET_PORT 端口上有一个监听 Socket...
   char  lpszServerAddr[20];
   BYTE  nField[4];
   m_ipCtrl.GetAddress(nField[0], nField[1], nField[2], nField[3]);
   sprintf(lpszServerAddr,"%d.%d.%d.%d",nField[0],nField[1],nField[2],nF
        ield[3]);
   if (m_pReceiver->Connect(lpszServerAddr, BASE_SOCKET_PORT))
   {
         m_bConnected = true;
         // 启动数据接收线程，即进入 CMediaSocketClient::ReceivingLoop 函数
         m_pReceiver->StartReceiving();
   }
   else
   {
         m_bConnected = false;
         AfxMessageBox("远程连接失败！");
   }
}
```

当用户要求服务器开始发送数据时，执行 CMediaClientDlg::OnButtonTrans 函数：

```
void CMediaClientDlg::OnButtonTrans()
{
   if (m_pReceiver)
   {
         char pData[sizeof(MSG_HEADER)];
         PMSG_HEADER pMsg = (PMSG_HEADER) pData;
         pMsg->nDataSize  = 0;
```

```
        pMsg->nMsgType    = DATA_REQUEST; // 指定一个负载类型（要求发送数据）
        m_pReceiver->Send(pData, sizeof(MSG_HEADER));
    }
}
```

18.4 问 题 会 诊

本章介绍的是笔者早些时候在网络上发布的一个开放源码。至今，这个开放源码已经引起了较多网友的关注。经过整理，这里再罗列一些这个源码在实际应用和扩展中的相关问题；希望这些问题总结，能够对大家有所帮助或者启示。

MediaClient 编译错误 1（Debug 环境下）

```
strmbasd.lib(wxutil.obj) : error LNK2005: "public: __thiscall
CCritSec::CCritSec(void)" (??0CCritSec@@QAE@XZ) already defined in
CFilterGraph.obj
strmbasd.lib(wxutil.obj) : error LNK2005: "public: __thiscall
CCritSec::~CCritSec(void)" (??1CCritSec@@QAE@XZ) already defined in
CFilterGraph.obj
strmbasd.lib(wxutil.obj) : error LNK2005: "public: void __thiscall
CCritSec::Lock(void)" (?Lock@CCritSec@@QAEXXZ) already defined in
CFilterGraph.obj
strmbasd.lib(wxutil.obj) : error LNK2005: "public: void __thiscall
CCritSec::Unlock(void)" (?Unlock@CCritSec@@QAEXXZ) already defined in
CFilterGraph.obj
```

解答：请确认 VC 的项目设置定义了宏 DEBUG，即选项 VC 菜单项 Project | Settings | C/C++，Category 选择 Preprocessor，然后查看 Preprocessor definitions 中是否有 DEBUG 的定义。另外，注意连接库的设置，DEBUG 版本连接 strmbasd.lib，Release 版本连接 strmbase.lib。

MediaClient 编译错误 2

```
asyncrdr.cpp
E:\DirectShowBook\SouceCodes\Chapter18\MpegNetwork\MediaClient\asyncrdr.
cpp(439) : fatal error C1010: unexpected end of file while looking for
precompiled header directive
asyncio.cpp
E:\DirectShowBook\SouceCodes\Chapter18\MpegNetwork\MediaClient\asyncio.c
pp(709) : fatal error C1010: unexpected end of file while looking for
precompiled header directive
```

解答：因为 asyncrdr.cpp 和 asyncio.cpp 两个文件是从 SDK 例子中直接拷贝过来的，它们没有用到预编译头文件 stdafx.h。应该确认如下的 VC 项目设置：选择 VC 菜单项 Project | Settings | C/C++，Category 选择 Precompiled Headers，然后选择 Automatic use of precompiled headers，并且 Through header 后面输入“stdafx.h”。

MediaClient 运行问题 1

客户端显示不出图像，只有 Video window 几个英文字母，但接收包的计数器仍然不

停地增加？

解答：出现这种问题，很有可能是 Filter Graph 没有构建成功。可以在 DEBUG 环境下跟踪客户端程序的执行，看究竟是哪一步出现问题。估计的原因，可能是在 Filter Graph 构建之前没有给 Source Filter 缓存一定数量的数据（可能不小心修改了原有程序的这种设计），也可能是服务器发送过来的数据本身就有问题（可以在客户端将接收到的数据直接写到一个文件中，然后看 GraphEdit 是否能够播放这个文件）。

MediaClient 运行问题 2

客户端播放的时候为何有间断？

解答：理想情况下（如网络流量保证畅通），客户端的播放应该是没有问题的。另外，可以考虑使用更加高效的数据缓存方案。

扩展问题 1

如何支持 MPEG2 的网络数据流？

解答：因为 DirectShow 没有提供 MPEG2 的视频解码 Filter，以及相应的一些音频解码 Filter（用于解码如 AC3、LPCM 等格式），所以要想将客户端扩展成能够播放 MPEG2 数据流，首要的条件是，提供第三方的上述几种 Filter。另外，需要修改 Source Filter 的输出媒体类型为（MEDIATYPE_Stream，MEDIASUBTYPE_MPEG2_PROGRAM）或（MEDIATYPE_Stream，MEDIASUBTYPE_MPEG2_TRANSPORT）。顺便说一下扩展支持 MPEG4 流的问题，因为很多采用 MPEG4 压缩算法的文件都保存为 AVI 格式；但 AVI 并不是一种流媒体，在 Source Filter 与 AVI Splitter 连接的时候，AVI Splitter 可能会要求读取 AVI 文件尾部的一些数据，这显然与流媒体的特性相悖。为了支持这种 AVI 文件的播放，可能要做出其他一些特殊处理，比如要求服务器首先发送一部分 AVI 文件尾部的数据等等。

扩展问题 2

如何支持多路同时显示？

解答：我们提供的开放源码限制了服务器与客户端是一对一的通信。在实际应用中，比如视频监控领域，服务器可能同时采集多路视频流，那么，如何在客户端同时播放多路视频流呢？首先要修改服务器程序：CListenSocket 类中有个成员变量 m_nLimitCount，默认情况下值为 1，即限制了最多只能与客户机建立一条 Socket 连接；实际应用中，可以指定一个更大的值。然后客户端程序也要修改，主要是为与服务器建立的各条连接分别创建一个 Filter Graph，然后将视频内容显示在不同的窗口中。当然，这种应用下的服务器与客户端还需要一种协议，能够让客户端指定服务器发送哪一路的视频流。

扩展问题 3

如何设计更好的应用层协议？

解答：多媒体数据的传送任务是很“繁重”的，也是很关键的。建议将数据传送和命令控制分开。数据传送可以使用可靠的 TCP 连接，而命令控制可以使用非面向连接的 UDP 数据报。

第 19 章　MPEG2 视频解码器

19.1　需 求 定 义

要使用 DirectShow 来播放一个 MPEG2 格式的文件，一般需要一个 MPEG2 Splitter，用于将 MPEG 流的音视频分离；需要一个 MPEG2 视频解码器，用于解码 MPEG2 格式的压缩视频；需要一系列相关的音频解码器，用于解码 AC3、LPCM 等格式的音频。遗憾的是，DirectShow 仅仅提供了 MPEG2 Splitter（或 MPEG2 Demultiplexer），而其他 Filter 一概空缺。本章就来解决这一问题。

目前市场上有很多第三方的 MPEG2 视频解码 Filter，也有一些开放源码的 MPEG2 解码相关的应用程序，比如 DVD2AVI。利用这些开放源码，不需要太多关于 MPEG2 的专业知识，也可以开发出一个 MPEG2 视频解码 Filter。

19.2　开放源码分析

下面，首先来分析 DVD2AVI 源码。这是一个 Win32 应用程序。可以把 MPEG2 解码部分的实现看成是"内核"，而把应用程序看成是基于"内核"之上的调用。这样，将"内核"分离出来，有助于以后 Filter 的开发。

研究别人的程序，最好的办法莫过于使用 VC 在 DEBUG 环境下跟踪；而且最好遵循应用程序设计的界面逻辑，以便更好地理解它的设计初衷。

提示

编译 DVD2AVI 的 DEBUG 版本，可能会出现如下错误：

```
Linking...
getpic.obj : error LNK2001: unresolved external symbol @SSE2MMX_IDCT@4
getpic.obj : error LNK2001: unresolved external symbol @SSEMMX_IDCT@4
getpic.obj : error LNK2001: unresolved external symbol @MMX_IDCT@4
Debug/DVD2AVI.exe : fatal error LNK1120: 3 unresolved externals
Error executing link.exe.
```

这是外部函数连接错误。可以暂时将 motion_compensation 函数（参见 getpic.c）中的 MMX_IDCT、SSEMMX_IDCT、SSE2MMX_IDCT 等函数调用注释掉，这样并不会影响对 DVD2AVI 程序流程的研究。记住，在编译 Release 版本的时候恢复这些函数调用！

运行 DVD2AVI 程序，并播放一个 MPEG2 文件，如图 19.1 所示。

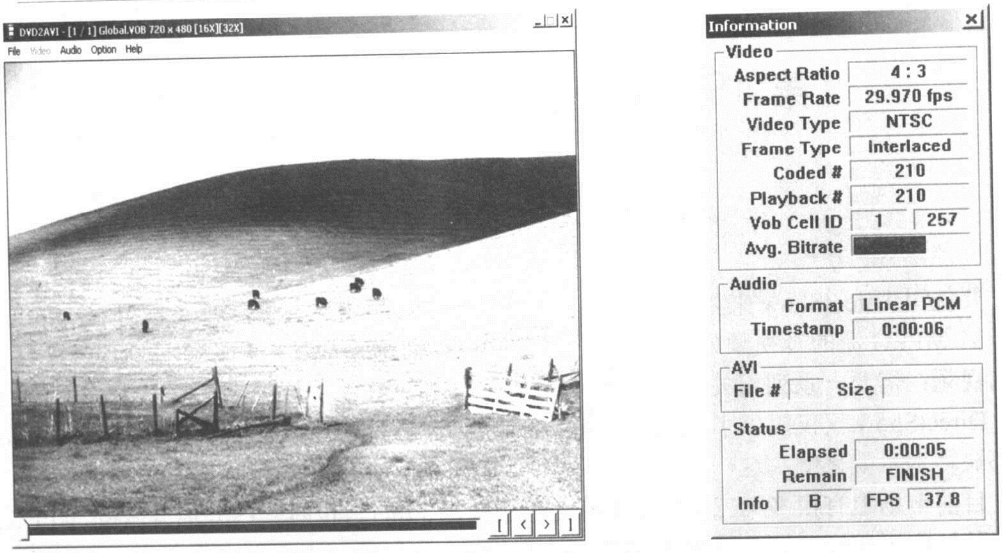

图 19.1 DVD2AVI 的程序界面

可以在执行菜单项 File | Open 之前，在回调函数 VideoList（媒体文件播放列表对话框的消息处理函数，参见 gui.cpp）的 case WM_INITDIALOG 下设置一个断点。可以看到，DVD2AVI 会调用 OpenVideoFile 函数弹出对话框，让用户选择媒体文件。而且选择的文件会在播放列表对话框中显示出来，如图 19.2 所示。

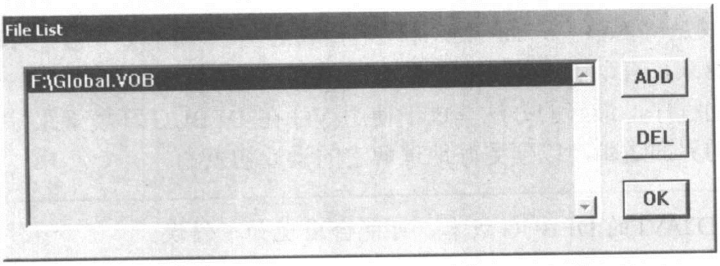

图 19.2 DVD2AVI 的媒体文件播放列表对话框

当单击 OK 按钮时，VideoList 函数就会响应 WM_COMMAND 消息的 IDOK 支路：

```
case IDOK:
case IDCANCEL:
   EndDialog(hVideoListDlg, 0);
   Recovery();   // 参数初始化
   if (File_Limit)
   {
         process.rightfile = File_Limit-1;
         process.rightlba = (int)(process.length[File_Limit-
                                1]/BUFFER_SIZE);
         process.end = process.total - BUFFER_SIZE;
         process.endfile = File_Limit - 1;
         process.endloc = (process.length[File_Limit-1]/BUFFER_SIZE -
                       1)*BUFFER_SIZE;
```

```
        process.locate = LOCATE_INIT;
        // 启动一个 MPEG2 的解码线程
        if (!threadId || WaitForSingleObject(hThread,
            INFINITE)==WAIT_OBJECT_0)
                hThread = CreateThread(NULL, 0, MPEG2Dec, 0, 0,
                            &threadId),
    }
    return true;
```

可以看到，DVD2AVI 在开始播放第一个媒体文件之前，首先调用了 Recovery 函数（参见 gui.cpp）进行系统初始化。另外还发现，DVD2AVI 使用了一个独立的工作者线程来完成 MPEG2 的解码工作。这个线程的执行函数为 MPEG2Dec（参见 mpeg2dec.c）：

```
DWORD WINAPI MPEG2Dec(LPVOID n)
{
    int i;
    // 标记变量初始化
    Pause_Flag = Stop_Flag = Start_Flag = Fault_Flag = false;
    Frame_Number = Second_Field = 0;
    Sound_Max = 1; Bitrate_Monitor = 0;
    // 音频属性初始化（可以忽略）
    for (i=0; i<CHANNEL; i++)
    {
            ZeroMemory(&ac3[i], sizeof(AC3Stream));
            ZeroMemory(&mpa[i], sizeof(RAWStream));
            ZeroMemory(&dts[i], sizeof(RAWStream));
    }
    ZeroMemory(&pcm, sizeof(struct PCMStream));
    ZeroMemory(&Channel, sizeof(Channel));
// MPEG 流处理类型（可以忽略）
    switch (process.locate)
    {
            case LOCATE_FORWARD:
                    process.startfile = process.file;
                    process.startloc = (process.lba + 2) * BUFFER_SIZE;
                    process.end = process.total - BUFFER_SIZE;
                    process.endfile = File_Limit - 1;
                    process.endloc = (process.length[File_Limit-
                                    1]/BUFFER_SIZE - 1) * BUFFER_SIZE;
                    break;
            case LOCATE_BACKWARD:
                    process.startfile = process.file;
                    process.startloc = (process.lba - 2) * BUFFER_SIZE;
                    process.end = process.total - BUFFER_SIZE;
                    process.endfile = File_Limit - 1;
                    process.endloc = (process.length[File_Limit-
                                    1]/BUFFER_SIZE - 1)*BUFFER_SIZE;
                    GOPBack();
                    break;
            case LOCATE_RIP:
```

```
            process.startfile = process.leftfile;
            process.startloc = process.leftlba * BUFFER_SIZE;
            process.endfile = process.rightfile;
            process.endloc = (process.rightlba - 1) * BUFFER_SIZE;
            process.run = 0;
            for (i=0; i<process.startfile; i++)
                  process.run += process.length[i];
            process.start = process.run + process.startloc;
            process.end = 0;
            for (i=0; i<process.endfile; i++)
                  process.end += process.length[i];
            process.end += process.endloc;
            break;
}
// 查询 MPEG2 流的序列头（Sequence Header）
if (!Check_Flag)
{
      int code;
      __int64 lfsr;
      // 一些界面元素初始化（可以忽略）
      if (KeyOp_Flag==KEY_OP)
      {
            lfsr = KeyOp(File_Limit, Infilename, hTrack);

            lfsr0 = (int)(lfsr>>32);
            lfsr1 = (int)(lfsr & 0xffffffff);
            if (lfsr)
            {
                  KeyOp_Flag = KEY_INPUT;
                  CheckMenuItem(hMenu, IDM_KEY_OFF, MF_UNCHECKED);
                  CheckMenuItem(hMenu, IDM_KEY_INPUT, MF_CHECKED);
                  CheckMenuItem(hMenu, IDM_KEY_OP, MF_UNCHECKED);
            }
            else
            {
                  KeyOp_Flag = KEY_OFF;
                  CheckMenuItem(hMenu, IDM_KEY_OFF, MF_CHECKED);
                  CheckMenuItem(hMenu, IDM_KEY_INPUT, MF_UNCHECKED);
                  CheckMenuItem(hMenu, IDM_KEY_OP, MF_UNCHECKED);
            }
      }
      File_Flag = 0;
      _lseeki64(Infile[0], 0, SEEK_SET);
      Initialize_Buffer();
      while (!Check_Flag)
      {
            // 检测 MPEG 的标识代码
            // 特征为连续三个字节为 0x000001
            next_start_code();
            // Get_Bits 函数（参见 getbit.h）可以从流中读出指定数量的数据
```

```
                // 读出 32 位标识代码进行判断
                code = Get_Bits(32);
                switch (code)
                {
                    // 检测到包头，说明这是一个系统流（可能同时包含音视频）
                    case PACK_START_CODE:       // 即 0x000001BA
                        SystemStream_Flag = true;
                        break;
                    case SEQUENCE_HEADER_CODE:
                            // 检测到序列头，即 0x000001B3
                        sequence_header();  // 读取 MPEG 流描述参数
                        InitialDecoder();   // 初始化解码器（很重要！）
                        Check_Flag = true;
                        break;
                }
            }
        }
        Frame_Rate = (FO_Flag==FO_FILM) ? frame_rate * 0.8f : frame_rate;
        // 将 MPEG2 解码后生成 AVI 文件的标记变量
        // 因为只需研究 DVD2AVI 的解码显示过程，所以这块代码可以忽略
        if (D2V_Flag)
        {
            // ...
        }
        File_Flag = process.startfile;
        _lseeki64(Infile[process.startfile],
                (process.startloc/BUFFER_SIZE)*BUFFER_SIZE, SEEK_SET);
// 缓冲初始化
        Initialize_Buffer();
        process.op = 0;
        // 首先在视频窗口中显示检测到的第一个 I 帧
        while (Get_Hdr() && picture_coding_type!=I_TYPE);
        Start_Flag = true;
        process.file = d2v_current.file;
        process.lba = d2v_current.lba;
        // 解码输出一帧图片（参见 getpic.c）
        // 如果当前程序处于初始状态，该函数在解码完一帧图片后会调用 ExitThread 函数
        // 终止线程的执行
        Decode_Picture();
        // 如果是连续播放，则继续执行以下函数
        // 找到一个 B 帧
        while (Get_Hdr() && picture_coding_type==B_TYPE);
        // 解码输出一帧图片（参见 getpic.c）
        Decode_Picture();
        process.op = process.mi = timeGetTime();
        // 连续播放的解码过程
        while (Get_Hdr())//定位到图片开始位置（Picture_Start_Code，即 0x00000100）
        {
            Decode_Picture();  // 解码输出一帧图片
            if (Stop_Flag)
```

```
            {
                    Fault_Flag = 99;
                    Write_Frame(NULL, d2v_current, 0);
            }
    }
    return 0;
}
```

下面再来看解码器初始化函数 InitialDecoder（实际上是一些系统参数的初始化，参见 mpeg2dec.c）：

```
static void InitialDecoder()
{
    int i, size;
    mb_width = (horizontal_size+15)/16;
    mb_height = progressive_sequence ? (vertical_size+15)/16 :
                2*((vertical_size+31)/32);
    // 解码后输出图像的宽、高
    Coded_Picture_Width = 16 * mb_width;
    Coded_Picture_Height = 16 * mb_height;
    Chroma_Width = (chroma_format==CHROMA444) ? Coded_Picture_Width :
                    Coded_Picture_Width>>1;
    Chroma_Height = (chroma_format!=CHROMA420) ? Coded_Picture_Height :
                    Coded_Picture_Height>>1;
    block_count = ChromaFormat[chroma_format];
    // 分配解码时用的帧缓存
    for (i=0; i<3; i++)
    {
            if (i==0)
                    size = Coded_Picture_Width * Coded_Picture_Height;
            else
                    size = Chroma_Width * Chroma_Height;
            backward_reference_frame[i] = (unsigned char*)_aligned_malloc
                                        (size, 64);
            forward_reference_frame[i] = (unsigned char*)_aligned_malloc
                                        (size, 64);
            auxframe[i] = (unsigned char*)_aligned_malloc(size, 64);
    }

    // 分配解码后输出的各分量缓存
    u422 = (unsigned char*)_aligned_malloc(Coded_Picture_Width*Coded_
       Picture_Height/2, 64);
    v422 = (unsigned char*)_aligned_malloc(Coded_Picture_Width*Coded_
       Picture_Height/2, 64);
    u444 = (unsigned char*)_aligned_malloc(Coded_Picture_Width*Coded_
       Picture_Height, 64);
    v444 = (unsigned char*)_aligned_malloc(Coded_Picture_Width*Coded_
       Picture_Height, 64);
    rgb24 = (unsigned char*)_aligned_malloc(Coded_Picture_Width*Coded_
       Picture_Height*3, 64);
```

```
    yuy2 = (unsigned char*)_aligned_malloc(Coded_Picture_Width*Coded_
        Picture_Height*2, 64);
    lum = (unsigned char*)_aligned_malloc(Coded_Picture_Width*Coded_
        Picture_Height, 64);
    CheckDirectDraw();
}
```

现在，应该对 DVD2AVI 的整个 MPEG2 解码过程比较清楚了。如果读者对 MPEG2 系统不是很熟悉，则完全可以不理会解码“内核”的实现。因为实际上 Filter 也只是一种应用框架。只模仿 DVD2AVI 应用程序对解码“内核”的使用过程，一样能够开发出自己的 MPEG2 视频解码 Filter。

19.3　Filter 设 计

要将 DVD2AVI 实现的 MPEG2 视频解码功能移植到一个 Filter 中，需要解决以下问题：

- 在 Filter 中使用一个独立的线程专门用于 MPEG2 的解码输出；
- 将 Filter 的输入 Pin 上传入的 Sample 数据组成一个连续的数据流，供解码线程读取；
- C 与 C++的混合编程。

要开发一个 Filter，选择一个合适的 Filter 父类是很重要的。综合考虑上述各种因素，决定采用 CSource 作为 Filter 的父类。因为 CSource 的输出 Pin（从 CSourceStream 类派生）已经实现了一个数据线程。在这个线程中，就可以控制整个解码过程。

另外，可以设计一个比较大的缓冲，来管理 MPEG2 流数据。当输入 Pin 有 Sample 输入时，直接将 Sample 数据放到这个缓冲中，等待解码线程读出；另一方面，解码线程也可以访问这个缓冲，并从中取出足够的数据以完成图像帧的解码。需要注意的是，应该对这个缓冲的操作进行线程同步（因为 Filter 的数据输入和数据输出使用的是不同的线程）。

由于 DVD2AVI 实现的 MPEG2 解码“内核”都是 C 代码，Filter 开发一般采用 C++。因此这里有一个混合编程的问题。解决方法是：设计一个 C++风格的应用控制类，将“内核”上可能被调用的 C 函数都在这个类中封装起来。于是，Filter 框架就只需与这个控制类打交道，从而降低了编程的复杂度，保持了一个清晰的程序结构。

19.4　Filter 编码

Filter 的注册名：“HQ MPEG-2 Video Decoder”；

Filter 类型：Transform Filter（有一个输入 Pin、一个输出 Pin）；

CLSID：CLSID_FilterMpeg2VD；

Merit 值：MERIT_PREFERRED；

功能描述：接收 MPEG2 的视频流输入，解码后输出 RGB24 或 YUY2 格式的图像帧序列。

使用 MPEG2 视频解码 Filter 构建的 Filter Graph 如图 19.3 所示。

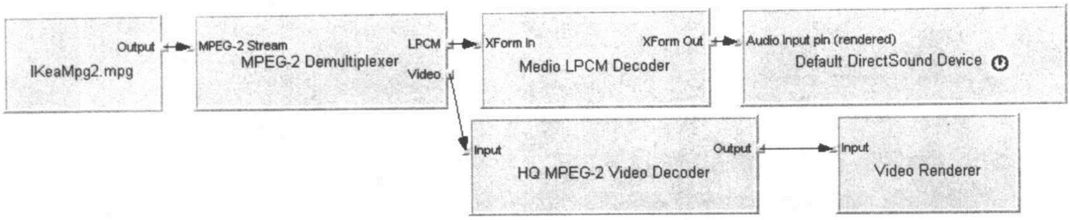

图 19.3　使用 HQ MPEG-2 Video Decoder 构建的 Filter Graph（1）

由于 DVD2AVI 解码“内核”直接输出的是 YUY2 格式，如图 19.3 的连接，让视频解码器输出 RGB24 的格式，实际上是在解码器内部进行了一次 YUY2 到 RGB24 的转换。由于 DVD2AVI 实现的这种色彩空间转换效率并不高，所以视频图像看起来会有间歇性的停顿。一种解决方法：可以在视频解码 Filter 和 Video Renderer 之间插入一个 AVI Decompressor，让解码器直接输出 YUY2 格式；这样，视频播放的观赏效果就要好很多。构建的 Filter Graph 如图 19.4 所示。

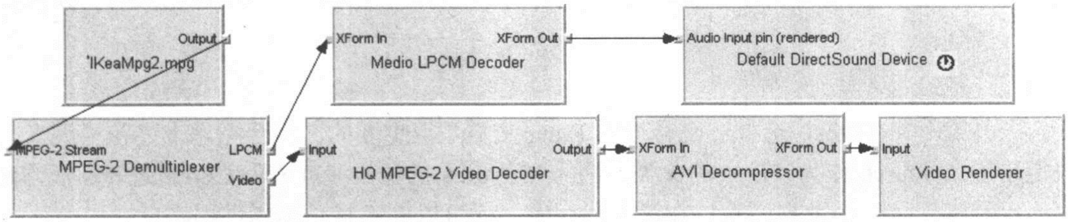

图 19.4　使用 HQ MPEG-2 Video Decoder 构建的 Filter Graph（2）

接下来，分析这个 MPEG2 视频解码 Filter 的实现。先看 Filter 框架：Filter 类名为 CFilterMpeg2VD，输入 Pin 类名为 CMpegInputPin，输出 Pin 类名为 CDecodedStream，它们的类继承关系如图 19.5 所示。

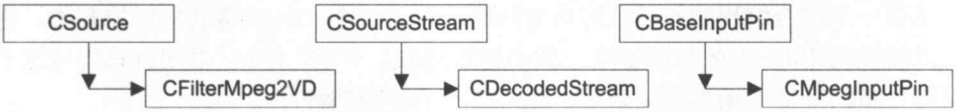

图 19.5　解码器 Filter 框架的类继承关系

CMpegInputPin 类重新实现了基类的几个函数：CheckMediaType、Receive、EndOfStream、BeginFlush、EndFlush、NewSegment、CompleteConnect 等。比如 CheckMediaType 函数的实现：

```
// 仅接收 MPEG2 视频的媒体类型
HRESULT CMpegInputPin::CheckMediaType(const CMediaType * mtIn)
{
    if ((mtIn->majortype == MEDIATYPE_Video ||
          mtIn->majortype == MEDIATYPE_MPEG2_PES) &&
          mtIn->subtype == MEDIASUBTYPE_MPEG2_VIDEO)
```

```
    {
        return NOERROR;
    }
    return E_FAIL;
}
```

其他的函数大多“委托”给了 Filter 去处理，比如 Receive 函数的实现：

```
STDMETHODIMP CMpegInputPin::Receive(IMediaSample *pSample)
{
    CAutoLock lck(&mDecodeFilter->m_csReceive);  // 数据线程同步
    ASSERT(pSample);

    // 首先调用父类的函数实现，以进行必要的状态检查
    HRESULT hr = CBaseInputPin::Receive(pSample);
    if (hr == S_OK)
    {
        // “委托”给 Filter 去接收数据
        hr = mDecodeFilter->Receive(pSample);
    }
    return hr;
}
```

CFilterMpeg2VD 类上定义了一个应用控制类 CMpegController 的对象实例，另外还实现了输入和输出 Pin 的“委托”函数。比如 Receive 函数的实现：

```
HRESULT CFilterMpeg2VD::Receive(IMediaSample *pSample)
{
    // 检查 Sample 的属性，如果不是媒体数据，则将它直接往下传送
    AM_SAMPLE2_PROPERTIES * const pProps = mMpegInputPin->SampleProps();
    if (pProps->dwStreamId != AM_STREAM_MEDIA)
    {
        return m_paStreams[0]->Deliver(pSample);
    }
    ASSERT(pSample);
    long lSourceSize = pSample->GetActualDataLength();
    BYTE * pSourceBuffer;
    pSample->GetPointer(&pSourceBuffer);
    // 将 MPEG2 视频数据直接交给控制类对象保存
    mMpegController.ReceiveMpeg(pSourceBuffer, lSourceSize);
    return NOERROR;
}
```

还有 CompleteConnect 函数，如下：

```
HRESULT CFilterMpeg2VD::CompleteConnect(PIN_DIRECTION inDirection, IPin
                                        * inReceivePin)
{
    if (inDirection == PINDIR_INPUT)   // 委托者是输入 Pin
    {
        CMediaType  mtIn = mMpegInputPin->CurrentMediaType();
        if (mtIn.formattype == FORMAT_MPEG2Video)
```

```
        {
                MPEG2VIDEOINFO * pFormat = (MPEG2VIDEOINFO *)
                                            mtIn.pbFormat;
                // 取得一些输入视频参数，如图像宽、高、帧率等
                mSampleDuration = pFormat->hdr.AvgTimePerFrame;
                mImageWidth     = pFormat->hdr.bmiHeader.biWidth;
                mImageHeight    = pFormat->hdr.bmiHeader.biHeight;
                // 初始化控制类对象中的解码器系统
                mMpegController.Uninitialize();
                mMpegController.Initialize();
                return S_OK;
        }
    }
    else  // 委托者是输出 Pin
    {
        // 判断输出 Pin 最终使用 YUY2 连接还是 RGB24 连接
        CMediaType  mtOut = OutputPin()->CurrentMediaType();
        int bitcount = 2;
        if (mtOut.subtype == MEDIASUBTYPE_YUY2)
        {
                bitcount = 2;
                mMpegController.SetOutputType(2);
        }
        else if (mtOut.subtype == MEDIASUBTYPE_RGB24)
        {
                bitcount = 3;
                mMpegController.SetOutputType(1);
        }
        // 计算解码后输出图像帧的大小
        mOutputImageSize = mImageWidth * mImageHeight * bitcount;
        mMpegController.SetOutputImageSize(mOutputImageSize);
        return S_OK;
    }
    return E_FAIL;
}
```

可以看到，CFilterMpeg2VD 类还实现了基类的几个函数，如 GetPinCount、GetPin、FindPin、Stop、Pause 等。这里，主要分析后两个函数的实现（因为 Filter 很容易进入死锁状态）：

```
STDMETHODIMP CFilterMpeg2VD::Stop()
{
    CAutoLock lck1(&m_cStateLock);
    if (m_State == State_Stopped)
    {
            return NOERROR;
    }
    // Succeed the Stop if we are not completely connected
    ASSERT(mMpegInputPin == NULL || m_paStreams[0] != NULL);
    if (mMpegInputPin == NULL || mMpegInputPin->IsConnected() == FALSE ||
```

```
        m_paStreams[0]->IsConnected() == FALSE)
    {
        m_State = State_Stopped;
        mEOSDelivered = FALSE;
        return NOERROR;
    }
    // 让控制类对象不再接收任何数据，以脱离循环等待状态
    mMpegController.FlushAllPending();
    // 析构输入 Pin 的 Sample 管理器，使输入 Pin 不能再接收数据
    mMpegInputPin->Inactive();
    // 获得输入数据线程的操作权
    CAutoLock lck2(&m_csReceive);
    // 析构输出 Pin 的 Sample 管理器，让输出线程也停止数据发送
    OutputPin()->BeginFlush();
    OutputPin()->Inactive();
    OutputPin()->EndFlush();
    // allow a class derived from CTransformFilter
    // to know about starting and stopping streaming
    HRESULT hr = StopStreaming();
    if (SUCCEEDED(hr))
    {
        // complete the state transition
        m_State = State_Stopped;
        mEOSDelivered = FALSE;
    }
    return hr;
}
STDMETHODIMP CFilterMpeg2VD::Pause()
{
    CAutoLock lck(&m_cStateLock);
    HRESULT hr = NOERROR;
    if (m_State == State_Paused)
    {
        // (This space left deliberately blank)
    }
    // If we have no input pin or it isn't yet connected then when we are
    // asked to pause we deliver an end of stream to the downstream
    // filter. This makes sure that it doesn't sit there forever waiting for
    // samples which we cannot ever deliver without an input connection.
    else if (mMpegInputPin == NULL || mMpegInputPin->IsConnected() ==
            FALSE)
    {
        if(m_paStreams[0]->IsConnected() && mEOSDelivered == FALSE)
        {
            m_paStreams[0]->DeliverEndOfStream();
            mEOSDelivered = TRUE;
        }
        m_State = State_Paused;
    }
    // We may have an input connection but no output connection
```

```
    // However, if we have an input pin we do have an output pin
    else if (m_paStreams[0]->IsConnected() == FALSE)
    {
            m_State = State_Paused;
    }
    else
    {
            if (m_State == State_Stopped)
            {
                    // allow a class derived from CTransformFilter
                    // to know about starting and stopping streaming
                    CAutoLock lck2(&m_csReceive);
                    hr = StartStreaming();
            }
            if (SUCCEEDED(hr))
            {
                    // 此时的输入线程不应该处于阻塞状态
                    mMpegController.FlushAllPending();
                    hr = CBaseFilter::Pause();
            }
    }
    return hr;
}
```

CDecodedStream 类主要控制整个 MPEG2 视频解码过程。下面主要来看 DoBufferProcessingLoop 函数的实现：

```
HRESULT CDecodedStream::DoBufferProcessingLoop(void)
{
    Command com;
    OnThreadStartPlay();

    BOOL   EOS_Flag = FALSE;
    do
    {
            while (!CheckRequest(&com))
            {
                    // 如果正在 Flushing，或者缓冲中没有数据流待解码
                    // 无需执行后续的解码流程
                    if (mFlushing || mMpegController->IsCacheEmpty() ||
                        EOS_Flag)
                    {
                            if (mDecodeFilter->mEOSReceived) // 输入数据接收完毕
                            {
                                    EOS_Flag = TRUE;
                                    mMpegController->EndEndOfStream();
                                    if (!mDecodeFilter->mEOSDelivered)
                                    {
                                            mDecodeFilter->mEOSDelivered = TRUE;
                                            DeliverEndOfStream();
                                    }
```

```
                }
                Sleep(1);
                continue;
            }
            // 正常的解码流程
            // 检查 MPEG 流的序列头（整个解码过程实际只进行一次）
            mMpegController->SequenceHeaderChecking();
            // 定位到图片头
            BOOL pass = mMpegController->LocatePictureHeader();
            if (pass)
            {
                // 解码出一帧图片
                pass = mMpegController->DecodeOnePicture();
            }
            if (pass)
            {
                // 取出输出 Pin 上一个空闲的 Sample
                IMediaSample *pSample;
                HRESULT hr = GetDeliveryBuffer(&pSample, NULL,
                                NULL, 0);
                if (FAILED(hr))
                {
                    Sleep(1);
                    continue;
                }
                // 往下发送 Sample
                hr = DeliverCurrentPicture(pSample);
                if (FAILED(hr) && mDecodeFilter->mEOSReceived)
                {
                    EOS_Flag = TRUE;
                    mMpegController->EndEndOfStream();
                    if (!mDecodeFilter->mEOSDelivered)
                    {
                        mDecodeFilter->mEOSDelivered = TRUE;
                        DeliverEndOfStream();
                    }
                }
            }
        }
        // For all commands sent to us there must be a Reply call!
        if (com == CMD_RUN || com == CMD_PAUSE)
        {
            Reply(NOERROR);
        }
        else if (com != CMD_STOP)
        {
            Reply((DWORD) E_UNEXPECTED);
            DbgLog((LOG_ERROR, 1, TEXT("Unexpected command!!!")));
        }
    } while (com != CMD_STOP);
```

```
    return S_FALSE;
}
```

其中，发送当前解码出来的图像帧 Sample 的函数 DeliverCurrentPicture 实现如下：

```
HRESULT CDecodedStream::DeliverCurrentPicture(IMediaSample * pSample)
{
    PBYTE   pOut;
    pSample->GetPointer(&pOut);
    // 从控制类对象中取出解码后的图像帧
    mMpegController->GetDecoded(pOut);
    // 设置 Sample 上一些属性，包括数据长度、时间戳等
    pSample->SetActualDataLength(mDecodeFilter->mOutputImageSize);
    ULONG    alreadySent = 0;
    {
           CAutoLock   lck(&mDataAccess);
           alreadySent  = mSamplesSent;
           mSamplesSent++;
    }
    LONGLONG   llStart = alreadySent;
    LONGLONG   llEnd   = alreadySent + 1;
    pSample->SetMediaTime(&llStart, &llEnd);
    REFERENCE_TIME rtStart = alreadySent * mDecodeFilter->mSample-
                             Duration;
    REFERENCE_TIME rtEnd = (alreadySent + 1) * mDecodeFilter->mSample
                           Duration;
    pSample->SetTime(&rtStart, &rtEnd);
    pSample->SetDiscontinuity(FALSE);
    pSample->SetPreroll(FALSE);
    pSample->SetSyncPoint(TRUE);
    // 将 Sample 发送出去
    HRESULT hr = Deliver(pSample);
    // 上述 Deliver 函数返回，表示下游的 Filter 已经完成当前 Sample 数据的处理
    // 可以调用 Sample 对象的 Release 函数将它收回
    pSample->Release();
    return hr;
}
```

CDecodedStream 类还实现了其他函数，比如 DecideBufferSize 决定输出 Sample 的内存大小，CheckMediaType 检查输出 Pin 连接时用的媒体类型，GetMediaType 提供输出 Pin 上的首选媒体类型，NonDelegatingQueryInterface 函数将随机定位操作往上一级 Filter 发送，Notify 将质量控制消息往上一级 Filter 发送等。这里就不再一一展开。

接下去，再来分析一下应用控制类 CMpegController 的实现。该类定义如下：

```
class CMpegController
{
private:
    long         mOutputImageSize;  // 输出图像帧大小
public:
    CMpegController();
```

```
    ~CMpegController();
public:
    bool Initialize(void);      // 初始化
    void Uninitialize(void);   // 反初始化
    void SetOutputType(int inType);     // 设置输出 Pin 输出的图像格式
    void SetOutputImageSize(long inImageSize);  // 设置输出图像的大小
    void BeginFlush(void);          // 开始数据的 Flushing
    void EndFlush(void);
    void BeginEndOfStream(void);   // 开始数据的 End-Of-Stream 处理
    void EndEndOfStream(void);
    void FlushAllPending(void);    // 使数据线程脱离循环等待状态
    // 输入 Pin 调用该函数接收 Sample 数据
    bool ReceiveMpeg(unsigned char * inData, long inLength);
    // 输出 Pin 调用该函数取出解码后的图像帧数据
    void GetDecoded(unsigned char * outPicture);
    BOOL IsCacheInputWaiting(void);  // 返回内部缓冲的状态
    BOOL IsCacheOutputWaiting(void);
    BOOL IsCacheEmpty(void);
    void SequenceHeaderChecking(void); // 检查 MPEG 流的序列头
    BOOL LocatePictureHeader(void);     // 定位 MPEG 流的图片头
    BOOL DecodeOnePicture(void);        // 解码出一帧图像
};
```

已知 CMpegController 类是与 DVD2AVI 解码“内核”直接相关的调用类。这里牵涉到一个 C 与 C++混合编程的问题。CMpegController.cpp 文件的开头如下定义：

```
#define GLOBAL
extern "C"
{
#include "global.h"
#include "getbit.h"
#include "smartcache.h"
}
extern "C" void InitSystem(void);
extern "C" void UninitSystem(void);
extern "C" void ResetSystem(void);
extern "C" void CheckSequenceHeader(void);
```

这些定义对于 C++类连接 C 函数符号至关重要。其他的成员函数实现，相对来说比较简单，都是调用相应的 C 函数。这些 C 函数包括 InitSystem、UninitSystem、InitSmartCache、ReleaseSmartCache、Get_Hdr、Decode_Picture 等，读者可以自行参考相关文件中的实现。

最后，需要提醒读者的是，由于 MPEG2 视频解码 Filter 使用的解码“内核”，使用了很多全局变量；当系统中存在多个这样的 Filter 一起工作时，这些全局变量会相互干扰，导致解码过程异常。因此，在同一时刻，应当只使用一个解码器实例。可以在 CFilterMpeg2VD 类的 CreateInstance 函数中加入这样的保护：

```
    CUnknown * WINAPI CFilterMpeg2VD::CreateInstance(LPUNKNOWN punk, HRESULT
*phr)
    {
```

```
    // ...
    // 如果解码器实例数超过 1，就不要让外部创建调用成功
    if (decoderInstances == 1)
    {
           *phr = E_FAIL;
           return NULL;
    }
    CFilterMpeg2VD *pNewObject = new CFilterMpeg2VD(NAME("Filter
                                  Mpeg2 VD"), punk, phr);
    if (pNewObject == NULL)
    {
            *phr = E_OUTOFMEMORY;
    }
    return pNewObject;

}
```

其中，decoderInstances 是一个 int 类型的静态全局变量，它的初始值为 0，每次在 CFilterMpeg2VD 类的构造函数中加 1，而在 CFilterMpeg2VD 类的析构函数中减 1。